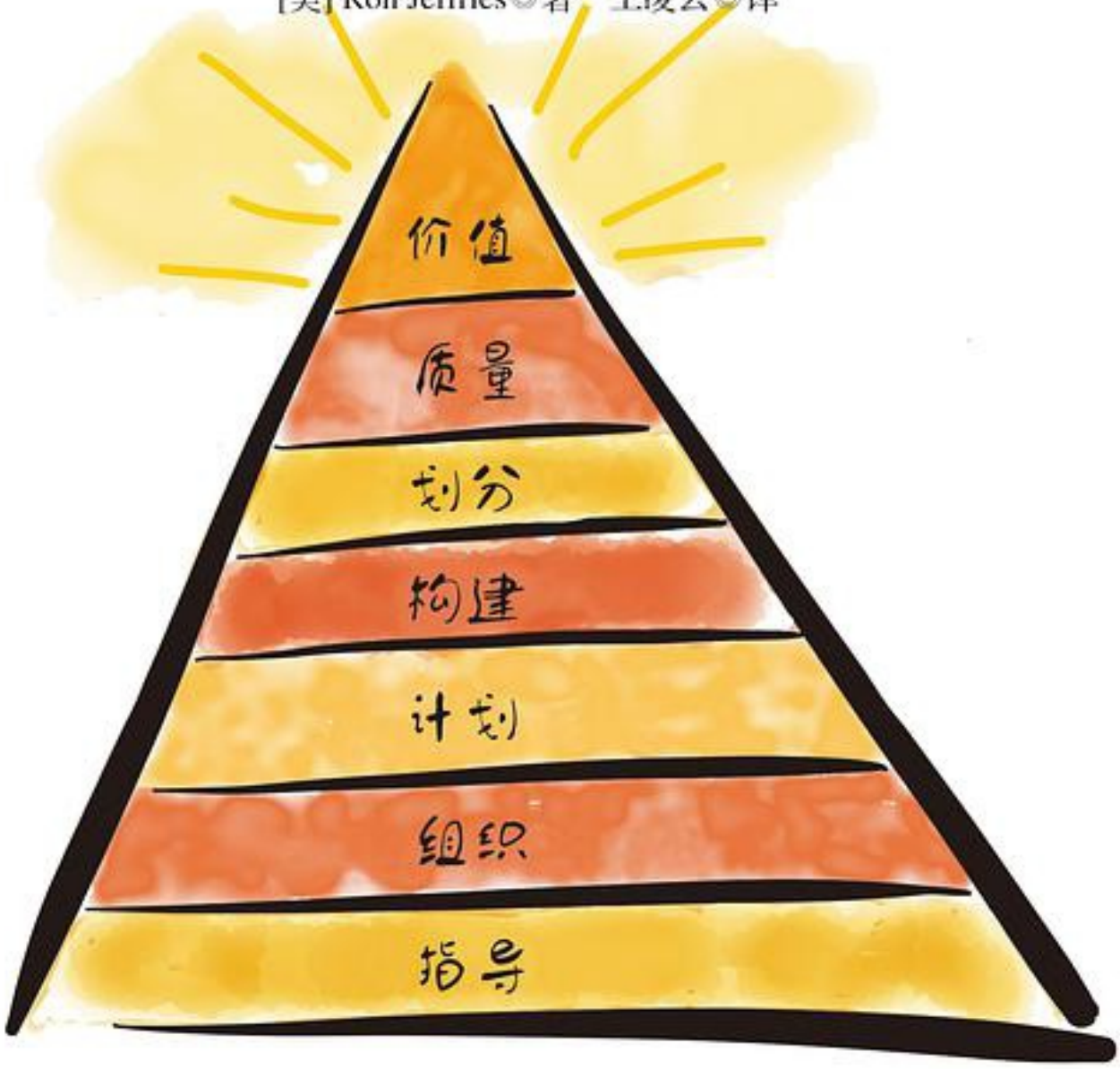
The Pragmatic Programmers

TURING 图灵程序设计丛书 全彩印刷

软件开发本质论

追求简约、体现价值、逐步构建

[美] Ron Jeffries 著　王凌云 译



中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

目录

封面

内容提要

文前

前言

致谢

引言

第一部分 价值的循环

- 第1章 寻找价值
- 第2章 价值就是那些我们想要的东西
- 第3章 根据功能特性可以指导得更好
- 第4章 根据功能特性组织团队
- 第5章 根据功能特性进行计划
- 第6章 根据功能特性构建产品
- 第7章 同时构建功能特性与基础
- 第8章 零缺陷与良好的设计
- 第9章 价值的完整循环

第二部分 说明与论述

- 第10章 价值是什么
- 第11章 如何衡量价值
- 第12章 是的，软件开发很难！
- 第13章 事情并非那么简单

第14章 组建强大的团队

第15章 使用五卡法进行初步的预测

第16章 自然软件开发的管理之道

第17章 监督员工更加努力地工作

第18章 能力是提高速度的前提

第19章 重构

第20章 敏捷方法

第21章 大规模敏捷

第22章 结论

封面

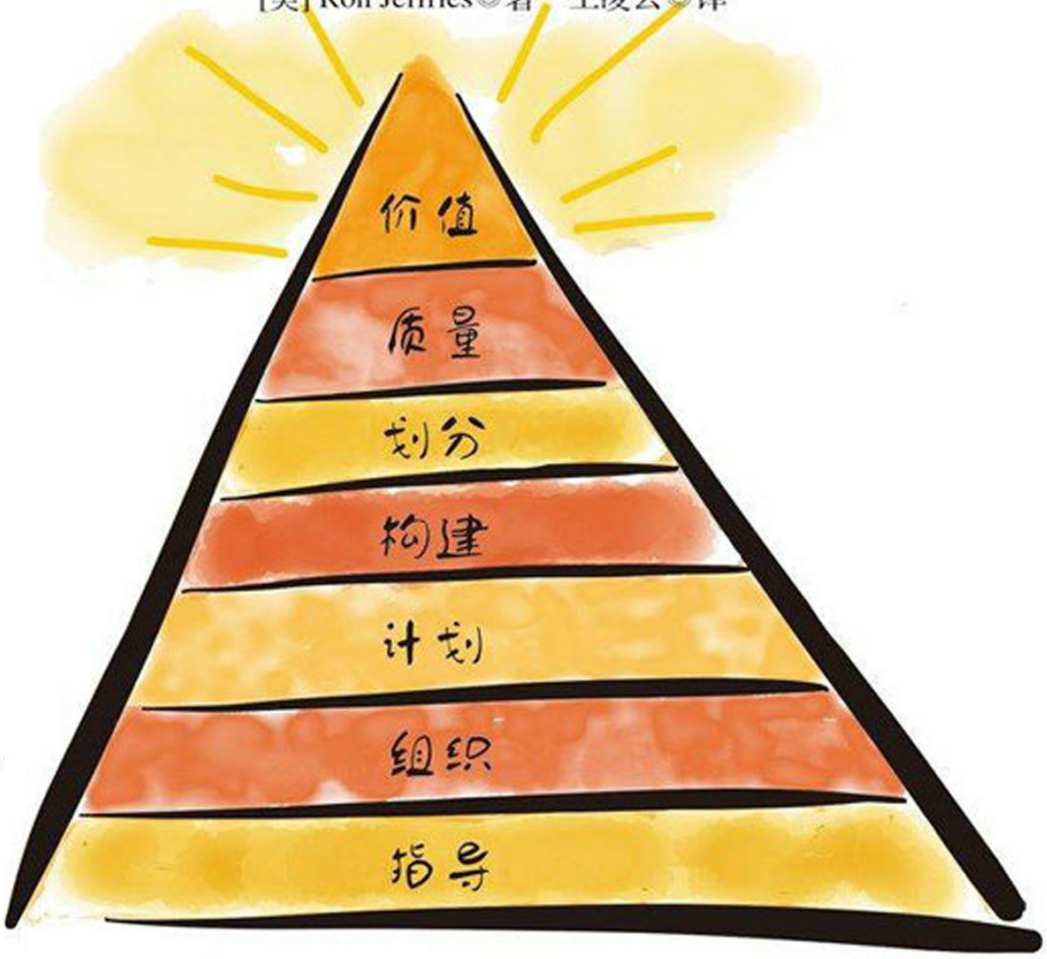
The Pragmatic Programmers

TURING 图灵程序设计丛书 全彩印刷

软件开发本质论

追求简约、体现价值、逐步构建

[美] Ron Jeffries 著　王凌云 译



中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

内容提要

本书以简单朴素的文字和生动活泼的手绘图向读者描绘软件开发的本质，并提出大量开放式问题，引领读者思考。作者勾画出一条敏捷开发的“自然之路”，指引软件开发者从复杂中找到简单的出路。本书分为两个部分。第一部分阐述价值的循环，并分析价值的本质、如何创造和交付价值，以及如何确保软件拥有良好的设计。第二部分针对读者可能产生的疑问进行解释，内容涉及如何衡量价值、如何组建强大的团队，以及是否应该实施大规模敏捷。

本书适合软件开发管理人员和所有软件开发者阅读。

本书由“ePUBw.COM”整理，ePUBw.COM 提供最新最全的优质电子书下载！！！

文前

“本书其实应该叫《写给CTO的专业软件开发指南》。对于每一位CTO、技术VP、软件产品总监、软件开发主管来说，本书都是必读书目。他们能够在书中针对几十年来一直困扰着同行的问题找到答案。本书写得十分简单明了，却阐述了人类试图解决的最复杂的问题之一，即如何管理构建高质量软件系统的开发团队。”

——Robert Martin

《敏捷宣言》起草人之一，Object Mentor公司创始人，人称“Bob大叔”

“快扔掉你身边那些充斥着时髦术语的书，开始阅读这本吧。本书带领我们回顾了软件开发的基础，总结出一套简单有效的软件开发流程，并向我们展示了软件开发的要素。如果你做得比书中所述更多，那就说明你想得太复杂了。”

——Jeff Langr

软件开发工程师与培训师，著有《C++程序设计实践与技巧：测试驱动开发》

“阅读本书就如同与作者罗恩相处了一个美妙的上午。”

——Chet Hendrickson

敏捷方法培训师与顾问，HendricksonXP公司

“我非常喜欢这本书。书中有大量手绘图，并配有清楚的解释，同时你可以立即尝试应用这些知识。这种阅读体验就像是喝着咖啡与罗恩促膝交流。”

——Daniel Steinberg

Dim Sum Thinking公司创始人

“在本书中，罗恩通过简单明了的语言和通俗易懂的手绘图，探讨了如何可靠高效地交付软件这一深奥的话题。本书不仅适合软件开发团队的成员阅读，同样也适合各种软件的客户和用户参考。”

——Bill Wake

Industrial Logic公司高级顾问

前言

我从事软件开发工作已经有半个多世纪了。在这些年里，我既获得过巨大的成功，也经历过彻底的失败。

这些年来，我一直在与他人讨论软件开发，并指导和教授软件开发。大部分时间里，我也在思考，试图弄明白为什么这样一件事情可以看起来十分简单，同时又十分复杂。如果你从事过软件开发工作，我想你可能也经常觉得这件事应该很简单，

但不知怎么却变得错综复杂。

托天时与地利之福，从一开始我就成为了敏捷开发运动的一员。它让我回归简单。

与软件开发中的很多最优秀的思想一样，现代的敏捷软件开发通过使工作变得更为简单，从而在提高开发效率的同时使我们可以更好地控制开发过程。敏捷很简单，概括地说，它只有四种价值和十二个原则。这能有多复杂呢？然而，它看起来似乎还是相当复杂。

像Scrum和极限编程这样的敏捷方法，其实也很简单。它们也只包括几种价值、几次会议，外加一些工件。这会有多复杂呢？然而，它们仍然很快就变得异常复杂起来。

这都是怎么啦？

我开始思索一种观察整个软件开发过程的方法。我正慢慢看到软件开发的概貌，它可以帮助我们保持简单。虽然其内部仍然比较复杂，但我希望这种概貌图可以帮助我们在发现自己身处杂草之中时重新回到简单的道路上。

软件开发涉及方方面面的内容，包括价值的确定、价值流的管理、相关工作的安排、计划的制订，以及软件的构建等。其中的每一个方面都必须以创造价值为中心，而且价值必须是可见的。只有这样，价值才便于我们进行指导和管理。而要实现这一点，我们需要从细节中抽身，并找出这种十分复杂的活动所蕴含的简单本质。

当思考问题时，我会围绕该问题的某个方面画一些画。我试着想出一些词语来帮助自己在下一次思考该问题时快速集中精力。我想通过图画来给自己一个不同的视

角。由于在绘画方面并不熟练，因此我的画很简单；我想通过这些画去掉复杂的内容，看看留下了什么。实际上，我是在以视觉方式向你展示我的思路。

本书试图从构建软件产品这一复杂的活动中找出某些简单的本质。我确信自己掌握了一些很不错的想法。然而，本书充其量也不过是在杂草丛生处清理出一条小道。请带上这些想法，并利用它们在一片混乱之中找到属于你自己的简单感觉。祝你好运！

致谢

从哪里开始……到哪里结束……

首先感谢我的父母，因为他们给予了我自由与信任，同时为我提供了丰富的藏书……

还要感谢Mary Marjorie修女，是她让我第一次领略到科学的奥妙；感谢Dansky先生，是他让我第一次品尝到热爱一门科学的滋味；同时我也要感谢耶稣会，是它向我展示了思考的价值，同时培养了我的时尚感。

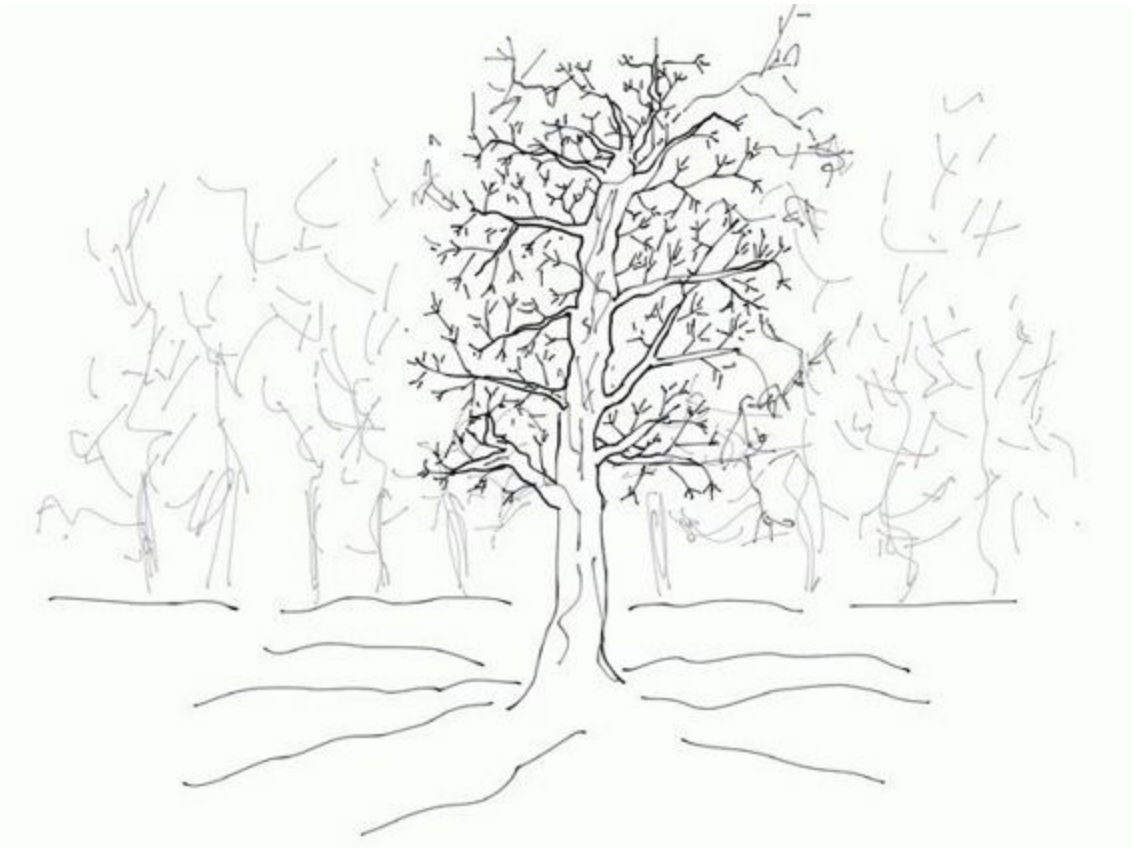
感谢Rick Camp，当我还是在街上游走的小青年时，是他的邀请使我成为了美国战略空军司令部的一名实习生；感谢Bill Rogers，是他带领我进入编程世界，并教我如何在编程的海洋中畅游。

同时也要感谢这些年来的同事：Charles Bair、Karen Dueweke、Steve Weiss、

Gene Somdahl、Rick Evarts、Mike McConnell、Jean Musinski、Jeanne Hernandez、Dorothy Lieffers、Don Devine……如果要列出所有曾给予我感动的同事，估计需要好几页纸。恕我不能一一列出。

感谢我在敏捷联盟中的伙伴、导师和同事：Ward Cunningham、Kent Beck、Chet Hendrickson、Ann Anderson、Bob Martin、Alistair Cockburn、Martin Fowler、Michael Feathers、Bob Koss、Brian Button、Brian Marick、Ken Schwaber、Jeff Sutherland、Ken Auer……同样，恕我不能一一列出我心存感激的所有人。

我也要感谢互联网和Twitter社区。看见我如此频繁地解释这些想法，大家肯定已经厌烦了。



感谢针对本书向我提供帮助的Bill Tozier和Laura Fisher。当然还要感谢Chet

Hendrickson，他认真细致地审阅了本书的所有内容并帮助定稿。如果本书还有什么错误的话，当然需要由他来负责。

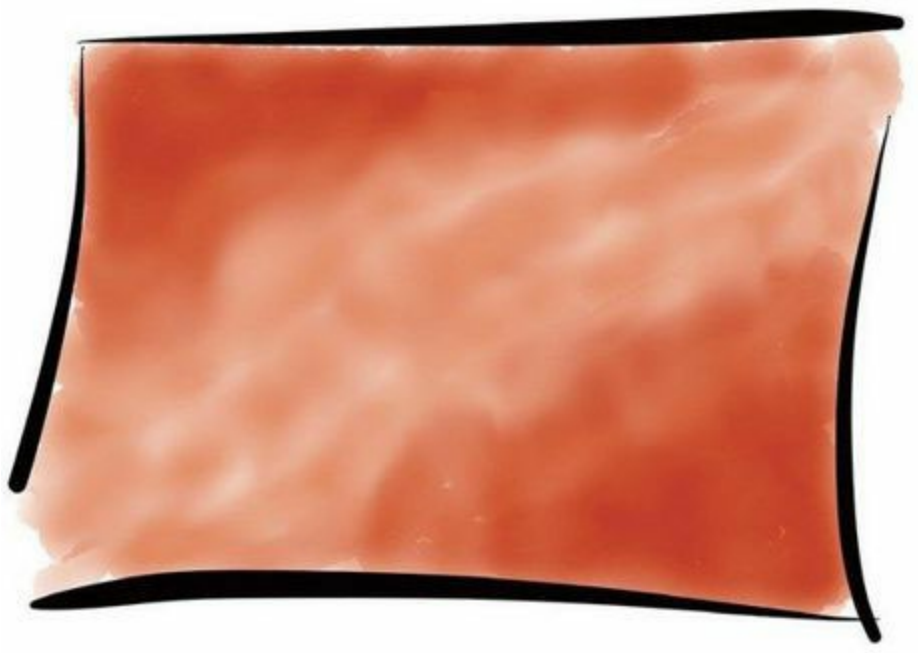
感谢The Pragmatic Programmers丛书的工作人员：Andy Hunt和Dave Thomas；Susannah Pfalzer，她能够很好地把握什么时候应该督促我，什么时候又应该站在幕后；Janet Furlow，她一直推动着本书的写作；当然还有本书的编辑，坚韧且耐心的Mike Swaine。如果没有他们，就不会有这本书。

感谢我的孩子Ron和Mike。我为他们感到骄傲，他们使我的生活充满了乐趣并且变得丰富多彩。

最后，我要衷心感谢我的妻子Ricia。如果没有她的话，我不知道这个世界上还有什么值得做的事情。感谢她一直以来对我无微不至的照顾。

感谢！

引言

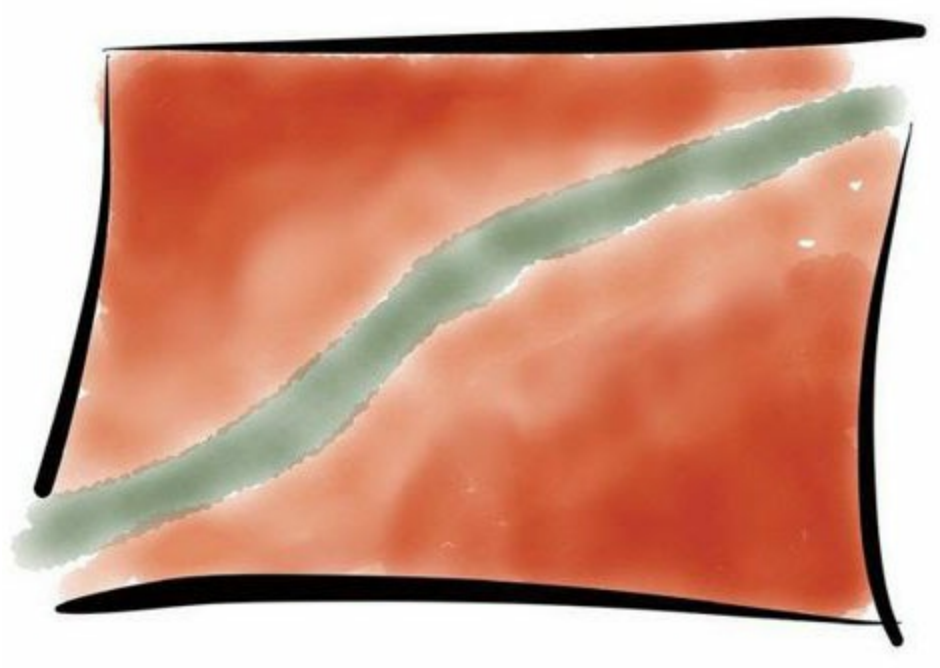


软件就像是熔岩

孩子们常常会玩一种叫作“地板就是熔岩”（The floor is lava）的游戏。在游戏中，你需要在不接触地板的情况下从一个地方挪到另一个地方，因为地板就是熔岩。如果踩到了熔岩，你就会被烫死；你会发出凄厉的号叫声，死相凄惨。因此，不要踩到熔岩。整个游戏中，你需要从沙发上跳到椅子上，然后从桌子的一头爬到另一头，最后跳进厨房这个避难所，因为这里的地板不是熔岩。

软件就像是熔岩，而且往往似乎并没有安全的落脚之地。更为糟糕的是，母亲大人不允许我们跳到家具上。面对这样的情况，真是遗憾。

那么，我们应该怎么做呢？构建软件时，我们似乎每天都踩在熔岩上。软件很复杂，并且会变得越来越复杂，而我们似乎注定要面对如此复杂的问题。



肯定有一条更好的路。

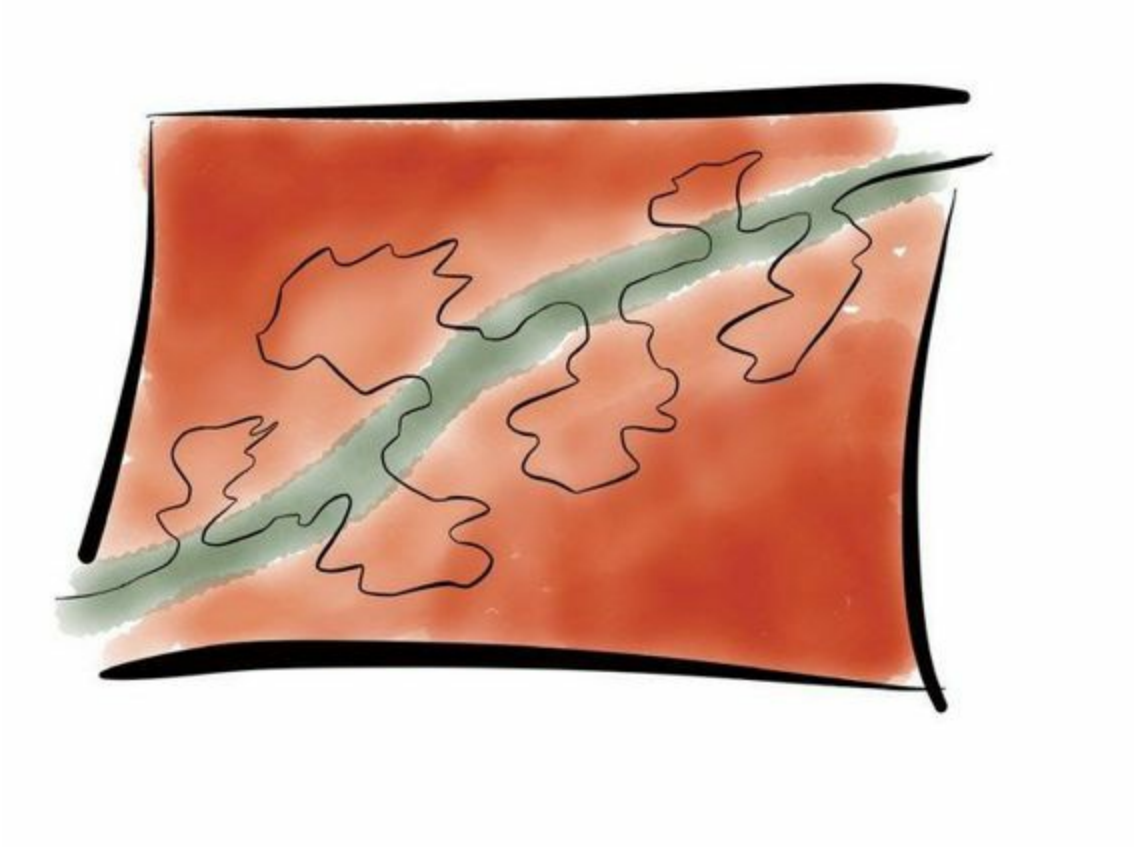
我们都有这样的感觉。我们确信，肯定存在一条不是由熔岩铺就的软件构建之路。上一次我们没能走上这条路，不过下一次……或者下下次……我们就能够走上这条康庄大道。

毫无疑问，结果是：下一次，我们踩到了更多的熔岩，在更凄惨的号叫声中死去。

然而，大多数人还是感受到了这样一个时刻：我们的脚并没有在灼烧，似乎这些熔岩之中夹杂着一些阴凉的草地。有时，我们很幸运地发现了这些草地。能够在这样阴凉的地方落脚，感觉真棒！

本书认为，存在于熔岩之中的不只是零星的草地，还有一条阴凉的绿色之路。或许，我们并不能时刻行走在这条路上，但更好地了解这条路则能使我们的项目之旅更加愉快。

我将这条路称为“自然之路”，因为我相信这条路是建立在如下这个简单的理念上的：尽早提供价值，经常提供价值。



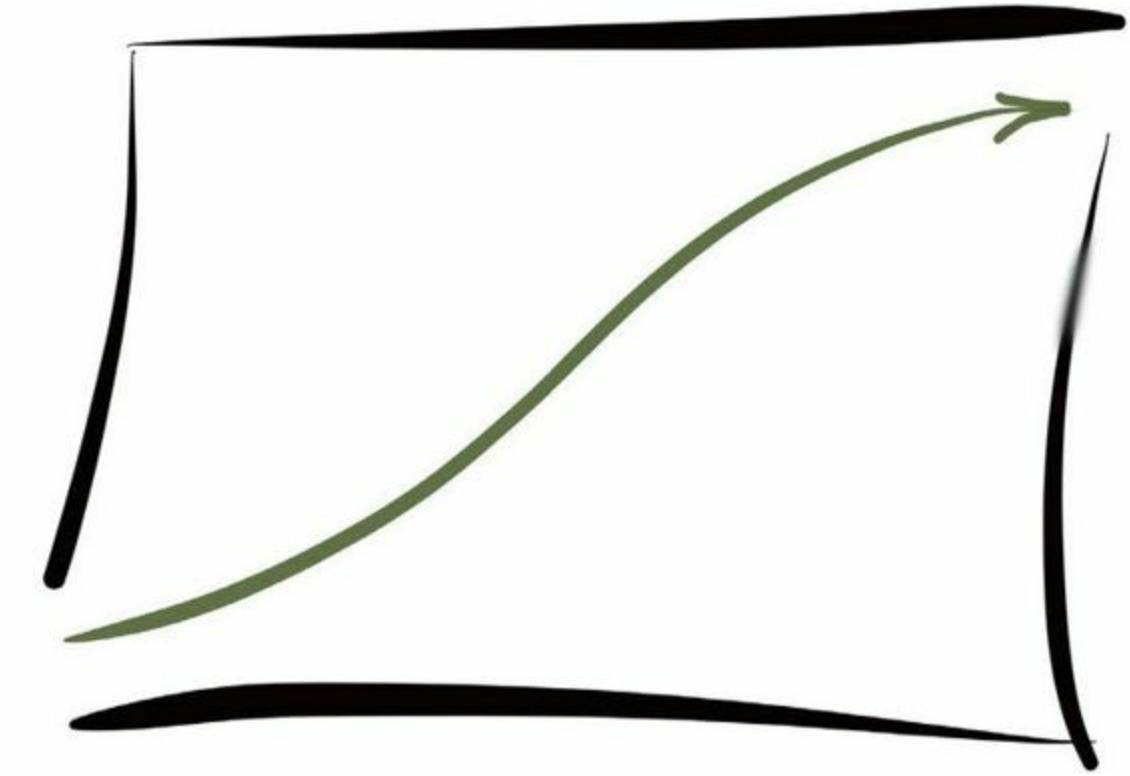
我们会偏离这条“自然之路”。

虽然与熔岩相比，我们更愿意走在草地上，不过似乎我们总是会踩到熔岩。（“熔岩”一词有时会有不同的含义，不过无论怎样，我们的处境都与踩在熔岩上相似。）

如果存在这样一条绿色之路（我希望能够向你证明它的确存在），我们还是会慢慢地偏离它。是的，确实如此。因此，在我向你描述这条路时，不要想象我会认为我们能够始终行走在这条路上，用我们满怀感激的双脚去抚慰这条路上的茵茵绿草，从此可以幸福地生活下去，再也不会遇到什么问题。我们不可能那样自在，或者说不会那么幸运。

我们所能做的，则是提醒自己，这样的路确实存在。当偏离这条路时，需要思考价值，同时也要想到这条自然之路。我们很有可能可以找到返回的路，即使不能重返

绿茵地，至少也会找到不那么炽热难熬的路。



自然之路

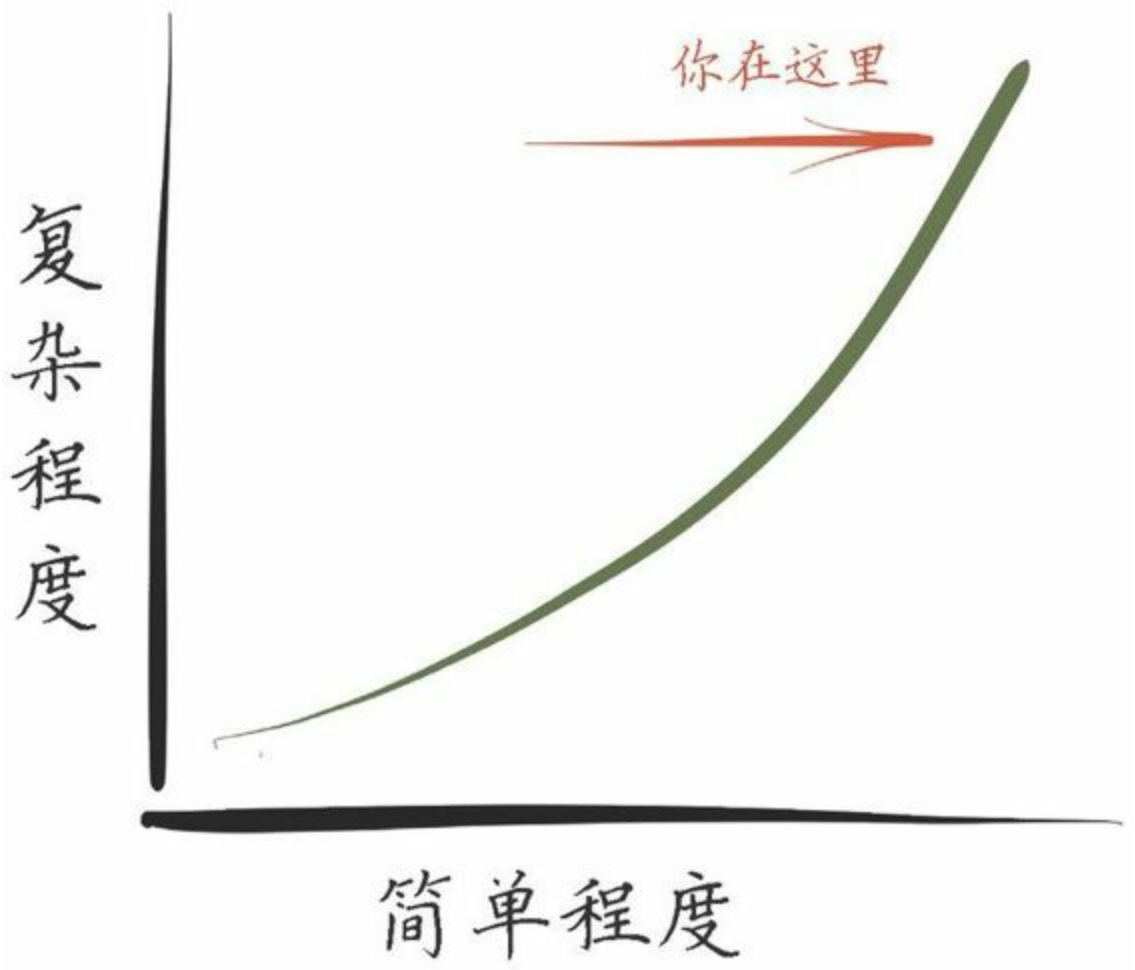
本书的故事很简单，即构建软件有一条自然之路，而且它对所有人都适用。

自然之路适用于最终用户，因为它能够更早为他们提供价值。

自然之路同样适用于公司，不仅因为它能够使投资更早地得到回报，并能够更快地提供重要的信息，同时还因为它能够使投资者在需要的时候及时调整方向。

自然之路也适用于管理人员。它能够使管理人员看到项目的真实情况，这样当需要采取措施时，管理人员能够有足够的时间采取相应的行动。同时，它能够使信息可见，免去挖掘信息的麻烦，从而减少管理中的问题。

自然之路甚至可以使开发人员的工作更加轻松。它为开发人员指明方向，从而使他们在必要时能自由地发挥自己的技能去构建客户需要的软件。



这里所描述的内容都很简单——不过并不容易。你需要思考这些理念，找出它们对你的价值，并学着去做本书探讨的事情。一直朝着简单的方向前进，将来你会为你曾经这样做而感到庆幸。

自然之路的确需要我们去思考，去学习，同时有所改变。我想，你将会在这里看到，走向自然之路并不一定是痛苦的。实际上，它可以很有趣。

让我们一起探索如何通过频繁提供可见价值来使软件开发变得更简单。我们将不会讨论事情是怎样的，而是会讨论如果我们尝试着去做的话它们可能是怎样的。

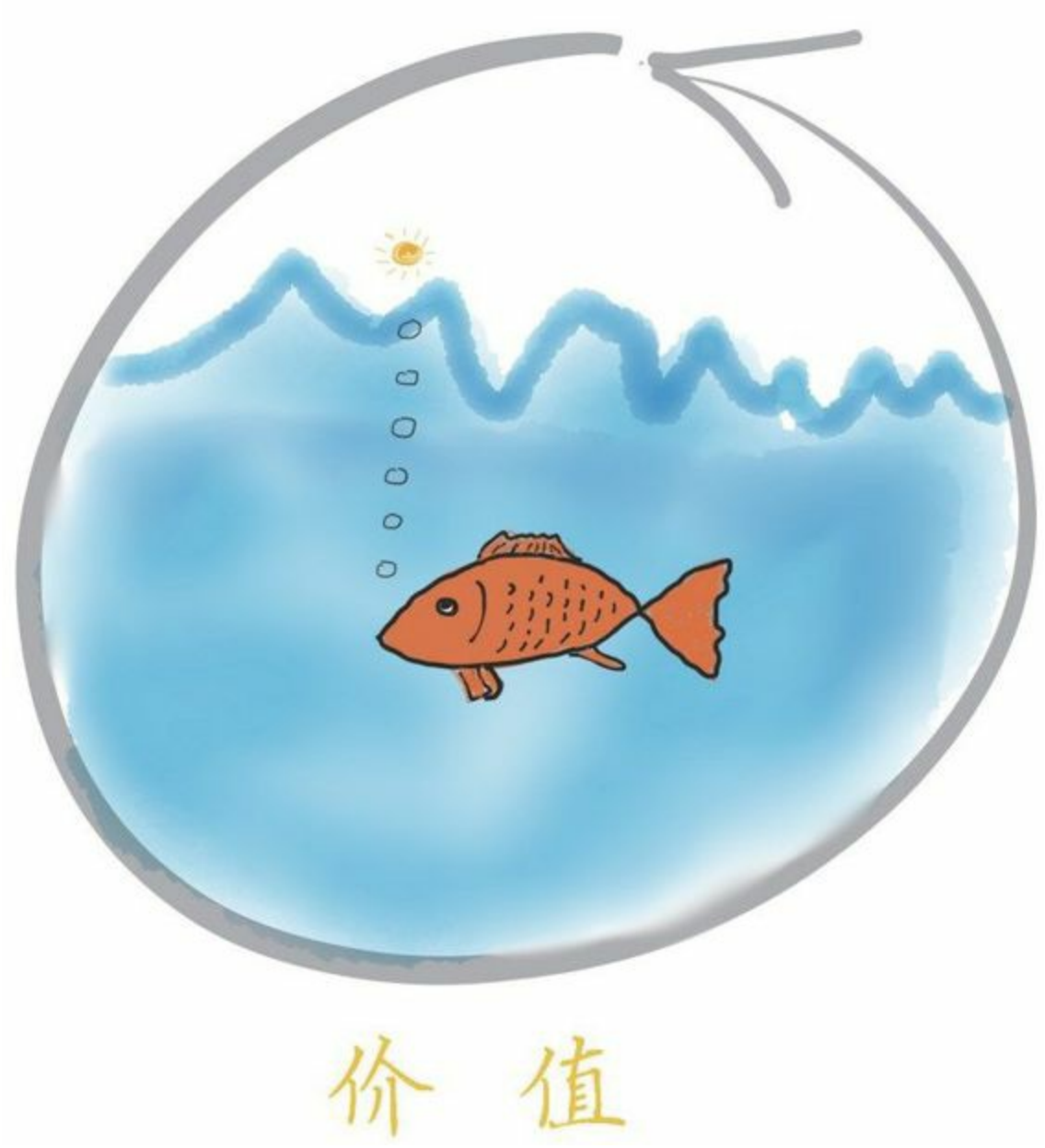
最后，在你继续阅读之前，我想给你个忠告。

借用喜剧演员艾迪·伊扎德（Eddie Izzard）在《死星餐厅》（Death Star Canteen）中的话：“这并不是一本关于到底要做什么的书！”

这不是一本关于诀窍的书，也不是一本关于某一种做事方法的书，这并不是我们的目的。我们需要思考事物的原理，并确保无论发生什么都能有所准备。很多方法都可以满足你的需要。我相信你能够找到这些方法，并会思考这些方法，然后做出选择。

第一部分 价值的循环

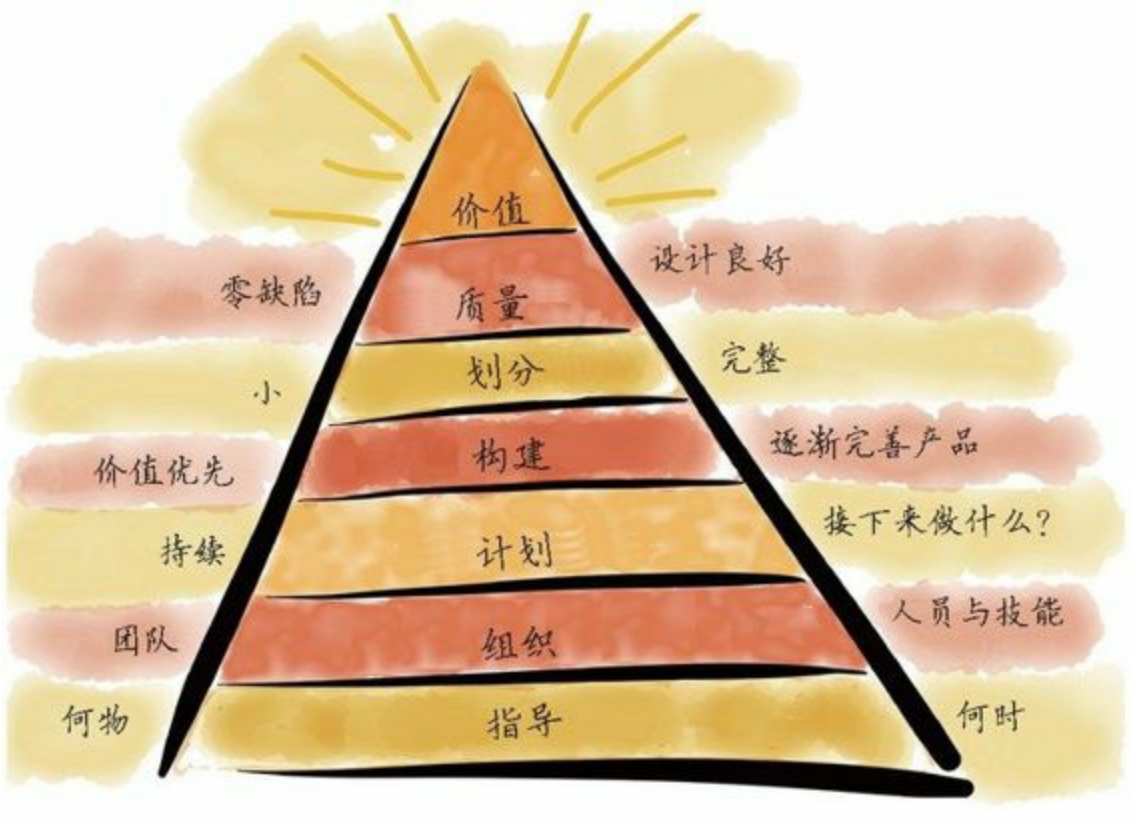
有时，为了去一个更好的地方，你不得不松开双手、双脚和尾巴，不再紧紧攀附着。当然，很可能你会掉到地上然后死去；但也有可能并不会这样，因为你天生就是做这个的料。



想要成功地开发软件是很困难的，而且这个过程始终很困难。然而，如果只是想要顺利、从容地经历这一过程，却是十分简单的。接下来，我们将一起讨论这一过程

所蕴含的简单本质。在讨论的时候，希望你能够勤于思考，而我会尽量少写一些内容以给你足够的思考空间。

第1章 寻找价值



上图展示了接下来我们要讨论的流程。让我们从价值开始，因为价值才是我们的工作重点。

价值：后面我们将会看到，所谓价值，就是“那些我们想要的东西”。价值有很多种，从金钱到笑容乃至生命，所有这些都是有价值的。第2章将更深入地探讨什么是价值。

我们会自下而上从金字塔的底部开始，在将产品划分为小块的基础上讨论如何指导、组织、计划和构建产品，同时重点关注产品的质量。我们将以此为基础，最终

创造价值。

指导：通过组建以创造价值为己任的团队来实现价值的创造，因此需要确保团队成员知道客户需要什么，以及客户留给我们的开发时间。我们通过观察实际构建出的产品来对团队的工作进行指导。

组织：为了更好地完成工作，需要对团队进行组织。我们需要围绕产品的功能特性来进行组织，因为这些功能特性可以使我们更好地计划，并更快地创造出价值。我们选贤任能，并帮助他们提高技能。

计划：根据所需功能特性的前后顺序来对其进行选择，以此来控制项目的进展。这样就能够及时创造价值。

构建：通过逐个实现功能特性来构建产品。这样就能够频繁地进行价值的交付，同时能够尽早、经常地看到项目的进展。

划分：将功能特性划分为小块，使每一块尽可能小，前提是它们仍然有价值。这样就能够尽早地构建出有用的产品，并在交付日期到来之前对产品进行优化与提升。我们时刻准备交付产品。

质量：采取必要的措施，以确保生产出来的产品设计优秀、品质精良。这样就能够不断、持续、永久地创造价值。

第2章 价值就是那些我们想要的东西

软件的价值是什么？

我们都想要价值。所谓价值，就是那些我们想要的东西。是的，价值就是那些我们想要的东西。在软件开发领域，通常我们通过交付功能来获得价值。当然，这些功能都是有价值的功能，也是我们想要的功能。

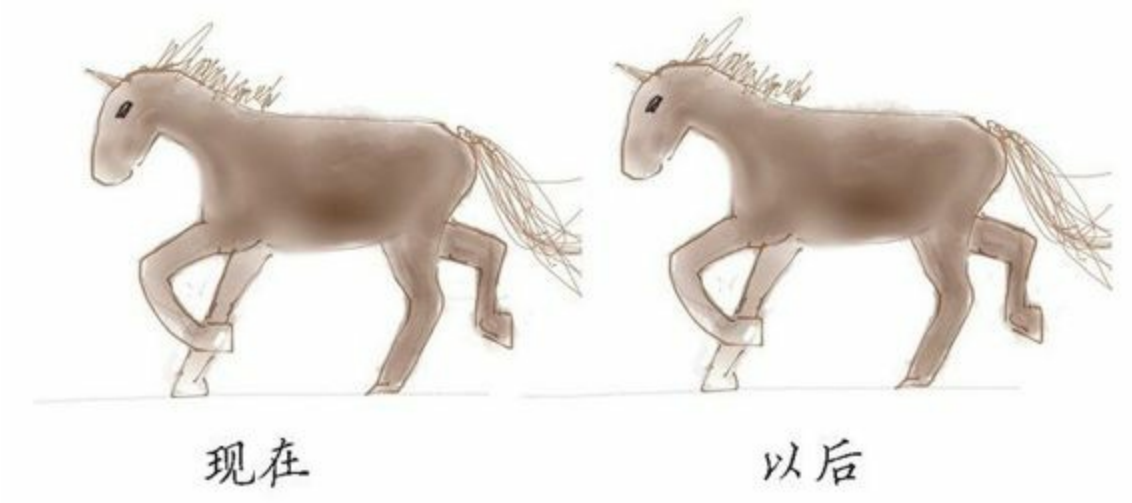
通常，软件的价值都与金钱有关，因为软件能够帮助人们节省时间或金钱。软件也能够帮助我们赚钱。当然，也有其他类型的价值，比如说软件能够使人们的生活更加便利。有时，软件甚至能够挽救人的生命。

总之，我认为价值就是那些我们想要的东西。或许我们很喜欢用数字来表示价值，然而并非必须这样做。当构建软件时，我们将做出很多选择，每一种选择都会带来一些我们认为有价值的东西。我们可能会选择信息、用户的快乐或者是挽救人的生命。我们会选择任何有意义的事，选择那些我们想要的东西。

随着这一工作的完成，我们会继续选择下一件想要的东西。我们会让研发团队以他们最快的速度、用最可靠的方式，将所选择的价值融入到软件中。当他们构建好软件之后，我们要确保得到了想要的东西，也就是说我们会检验软件的价值。

为了检验价值，我们会说：“请演示一下软件。”

那么，你的项目会为用户提供什么样的价值呢？它又会为你的公司带来什么样的价值呢？此外，它还能够为你所在的团队（当然也包括你）带来什么样的价值呢？



当软件发布时，它的价值才能够体现。

只有当软件发布并被实际使用时，项目才实现了价值的交付。如果要一直等到所有的工作完成，那么通常要过很长一段时间才能够获得价值。因此，我们需要找到一种方法尽早地交付价值。

假设我们想现在就拥有如图所示的小马，而不是以后，但现实是我们并不能够马上就创造出所有的一切。我们心中有许多想要实现的功能特性，但实现这些功能特性需要花费一定的时间。想要的功能特性越多，需要花费的时间就越长。

那么，尽早交付会有什么好处呢？公司又会如何受益呢？团队以及你我个人又会如

何呢？

如果软件的某个有价值的部分能够比其他部分更早交付，会是什么情况？

每个产品都由很多部分组成。我们将这些组成部分称作功能特性（feature），或者最小可市场化功能特性（minimum marketable feature，MMF）；也可以称它们为方面（aspect）、功能（function）或者能力（capability）。每一个大的功能特性又都由更小的部分组成，这些满是细节的组成部分使整个大的功能特性更完整、更有用，或者说更完善。

需要记住的是，大部分用户并不会使用产品的每一个功能特性，这里会有一种人们通常所说的80/20法则存在。每个人想要的功能特性可能都不同，但是没有人想要所有的功能特性。即使是那些你最熟悉、用得最多的产品，你也很有可能只是使用了其中一部分的功能特性。

推出更小的产品有意义吗？

既然大部分用户并不会使用所有的功能特性，那么更小的功能特性集合就能够提供真正的价值，而且推出这样的产品也会更快。有时，我们会认为必须实现所有的功能特性。面对现实吧：如果你曾经经历过很多软件项目，很可能会知道，在计划的时间里是不可能得到所有想要的功能特性的。我们从来都没有做到过。

我们可以捶胸顿足，要求立刻有一匹小马，或者也可以表现得更像管理者，引导软件项目获得最好的结果。很有可能会发生下面这种情况：软件的一部分功能特性能够更早地服务于用户并产生价值。因此，我们需要找出并首先推出这些功能特性。这样就会获得成功。

在第一次发布后，需要继续跟进产品的其他功能特性，否则最终的产品在整个生命期内的价值就会缩水。因此，通常我们都会有多次发布版本的计划。

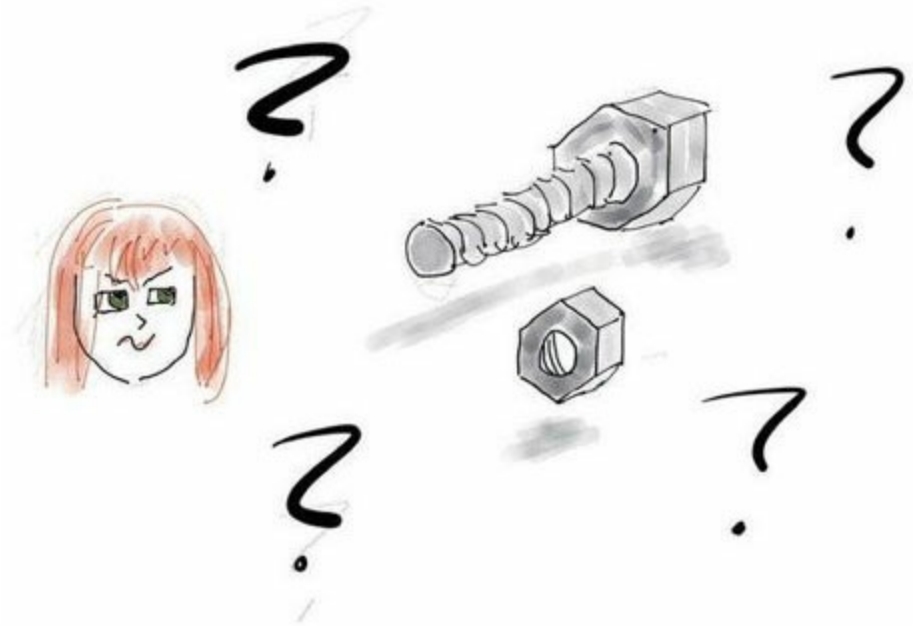
但是，也有可能只发布了第一版，然后就停止了。那么，在什么情况下这是最好的做法呢？你能够想出多少种理由呢？

只交付一部分功能特性就不再发布？

如果只发布一次的话，我们将会更早获得回报。但是，与发布完整的产品相比，这样做所获得的回报可能会少得多，即使发布完整的产品在时间上要迟很多。真的如此吗？

信息同样也有价值。有时，我们能得到的最重要的信息就是，自己正在做一件错误的事情。要想知道方向是否正确，一个好的方法就是尽早发布产品的小版本。这样如果失败了，就能够以较小的代价转变方向。

更常见的情况是，我们的想法的确很不错。有了不错的想法，应该从哪些功能特性入手呢？产品应该怎样一次只发布一部分功能特性呢？划分出的功能特性又是什么样的呢？



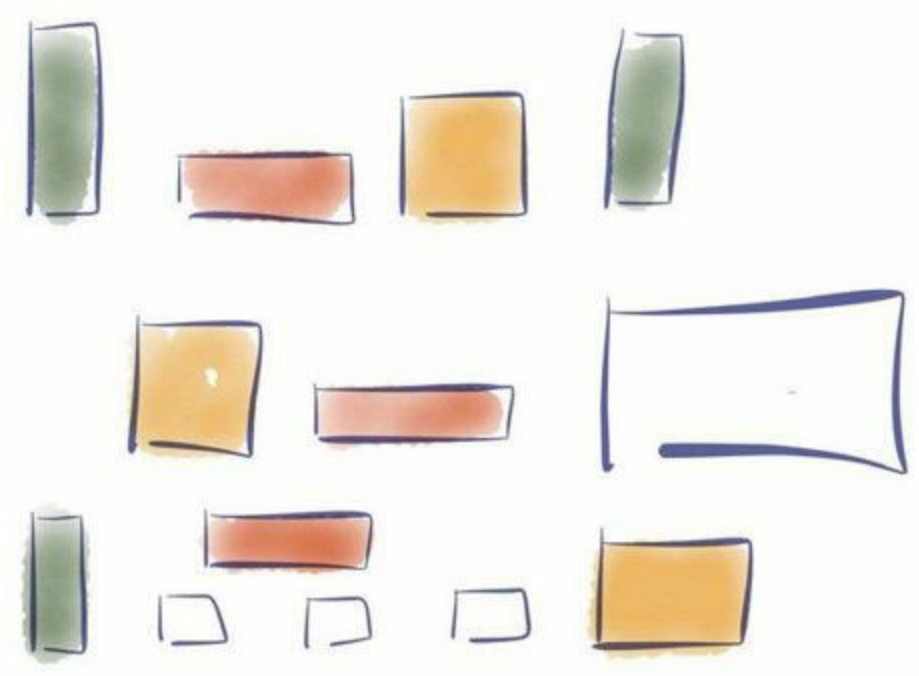
我们必须看到并且理解产品的各个部分。

研发团队整天从事着那些只对他们有意义、神秘的技术工作，这样做是不够的。

我们需要引导团队构建不仅对我们有意义，同时对用户也有意义的功能特性。这样的功能特性通常被称为最小可市场化功能特性。事实上，与通常的最小可市场化功能特性相比，在更细的粒度上提供商业方向更能使我们受益。

我们称这些部分为功能特性。当说“请演示一下软件”时，我们想要看见的是那些我们想要并且理解的功能特性。

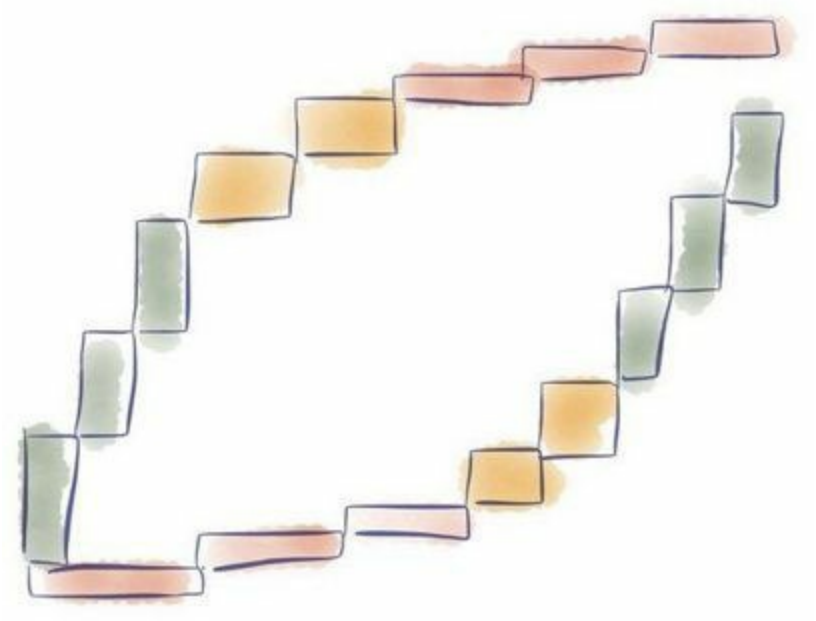
回顾以前的项目，那些你希望能够更早发布的功能特性是什么？你为什么希望这些功能特性能够更早发布？又有哪些功能特性应该是不同的呢？是否还有一些功能特性根本就不应该实现？



根据功能特性划分价值。

可能实现的每一个功能特性都会为整个产品增加一些价值；同时，实现每一个功能特性都需要花费一定的时间。虽然并不能够确切地知道每个功能特性有多大的价值或者需要花费多少时间，但是我们仍然可以很好地感知应该做些什么。

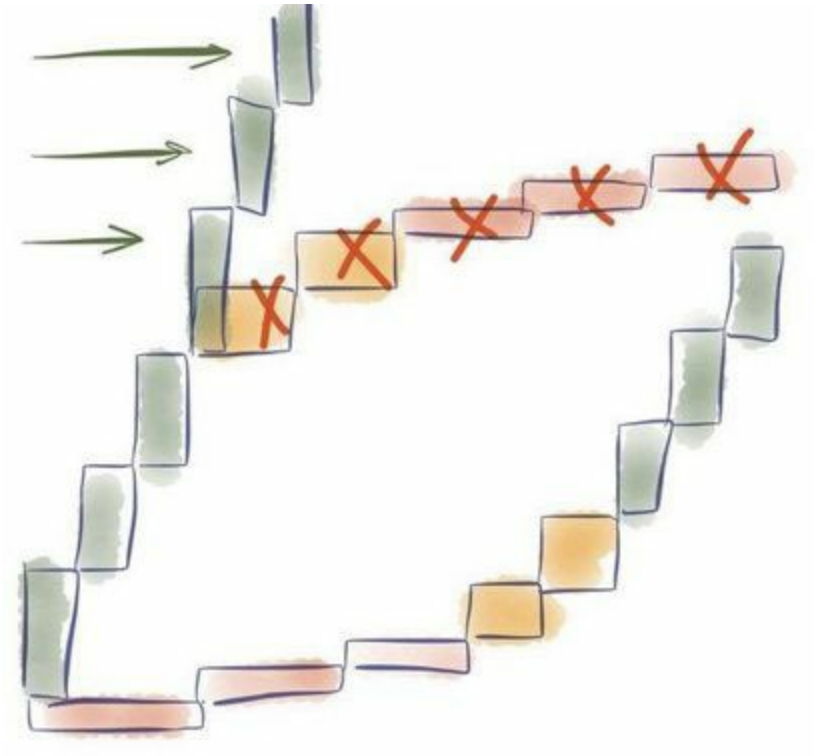
假设每个功能特性的高度代表它的价值，而宽度代表它的成本。那么，哪些功能特性是我们应该首先实现的？哪些又是可以推迟到以后去实现的？这样，我们可以一目了然，不是吗？



价值的增长取决于我们选择做什么。

如上图所示，如果我们选择首先完成高价值、低成本的功能特性，并将低价值、高成本的功能特性推迟到以后去完成，就能够看出价值增长的巨大差异。注意，图中所示价值差异的比值最大也只是3：1。然而，在大多数产品中，最好的想法要比最坏的想法好几十倍，甚至更多。如果是这种情况的话，则结果很难在这一页上显示。

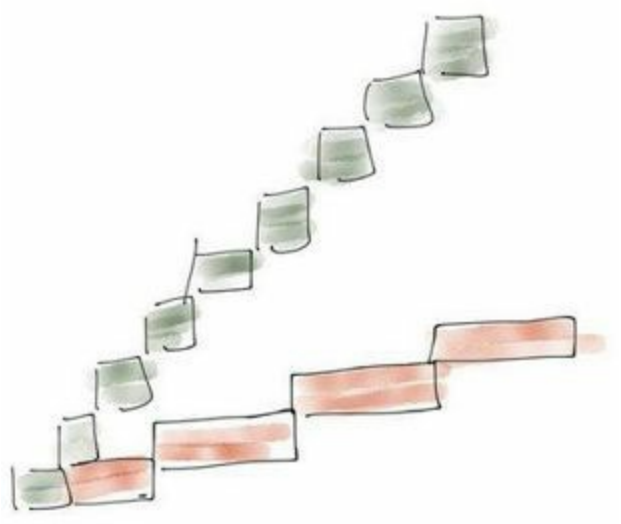
一些被推迟实现的功能特性看起来相当无趣。如果我们去实现一些不同的、更有价值的功能特性，甚至是在另外的产品上这样做，又会有什么情况发生呢？



我们甚至可以将投资转移到新的产品上！

当我们频繁地优先发布价值最高的功能特性时，很快就会发生下面这种情况：下一个要发布的功能特性并不值得我们花费这么多时间与金钱去实现。发生这种情况，是一件好事。通过投资新产品，通常我们都能做得更好。

那么，我们想要做什么样的新产品呢？又会有谁因为产品变更而受到负面的影响呢？怎样才能使产品变更对所有人来说都是好事呢？我们是否能够专注于系列产品，而不是回报率一直在降低的不同产品？我们能够提供更多软件并且同时提供更多价值吗？

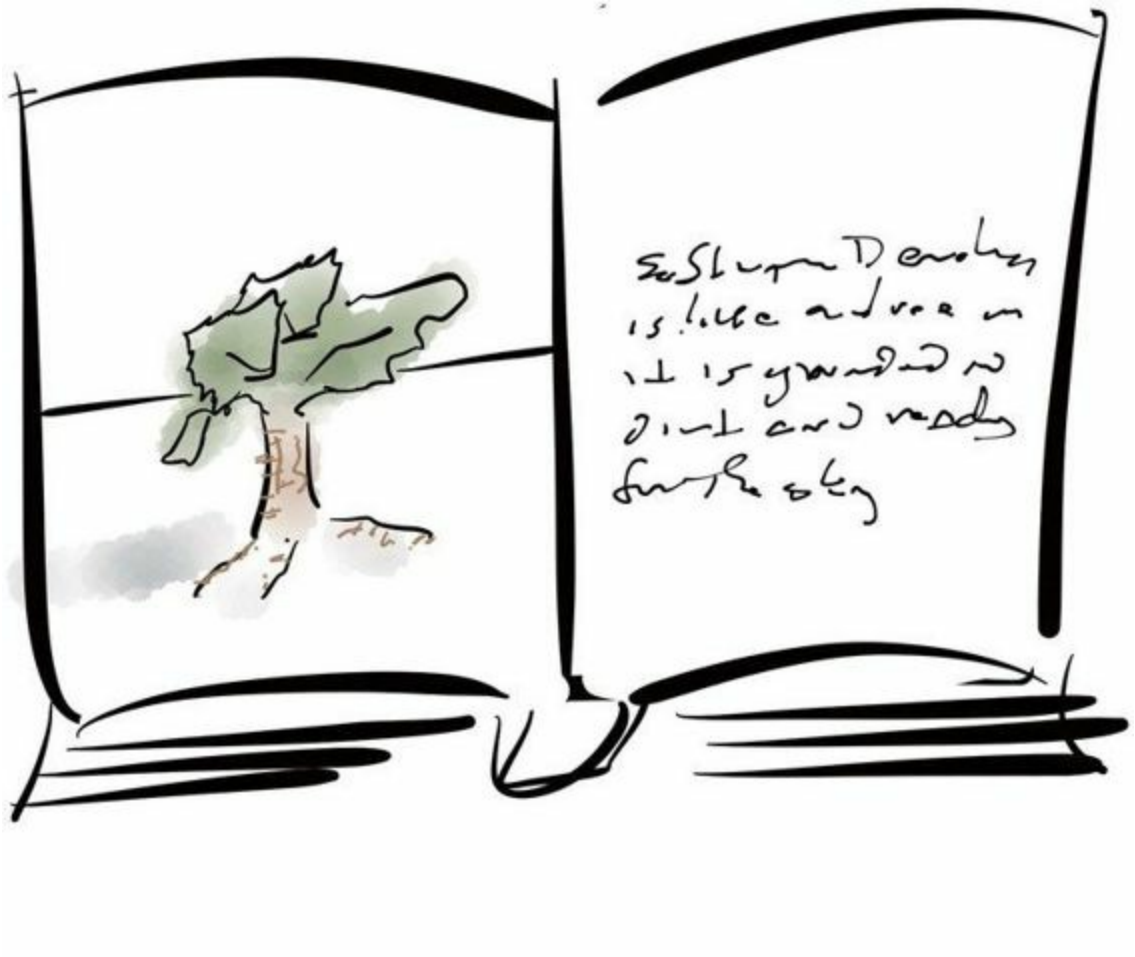


价值的最大化在于频繁交付小的、以价值为中心的功能特性。

好了，我们看到，如果能够这样做的话，小的功能特性可以更早地实现价值的交付。下面，我们来考虑项目的管理问题。更小且可见的结果是否有助于项目管理？它们又是否会有碍于管理？

我们的团队又是怎样的呢？他们是按这种工作方式组织起来的吗？团队拥有所需要的人员、所需要的技能，以及所需要的帮助吗？请继续阅读本书后面的内容，我们会逐一讨论所有这些问题。

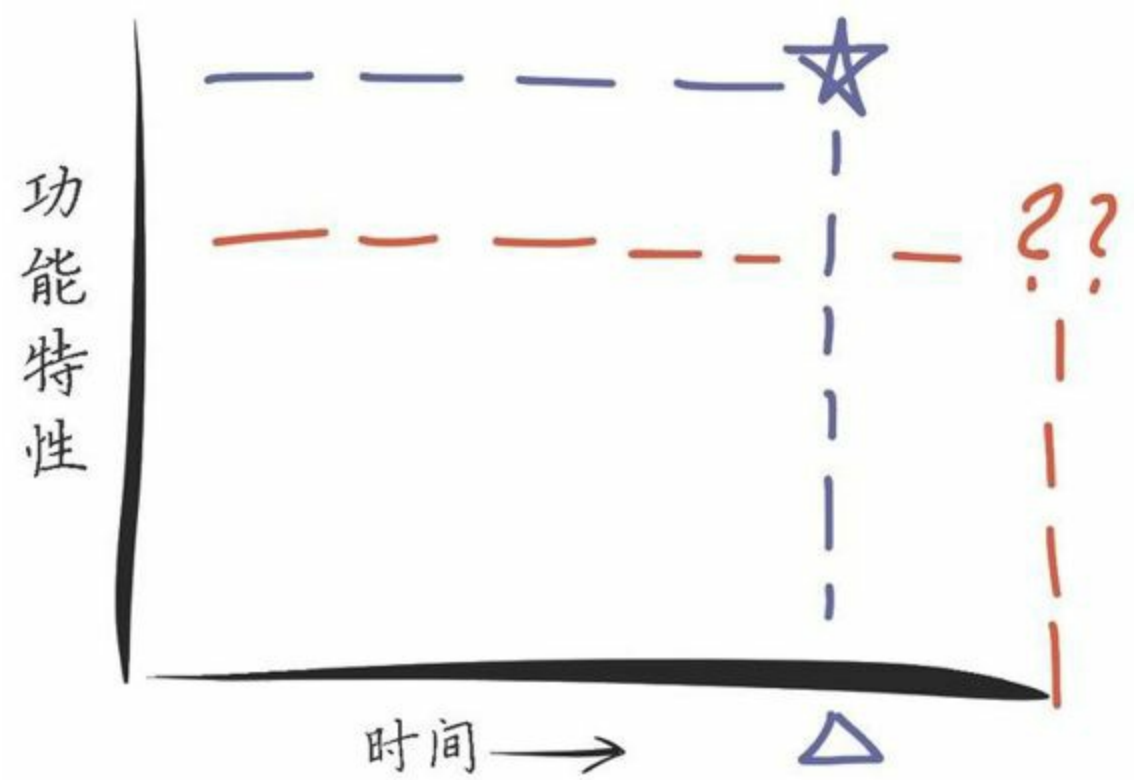
需要记住的是，以逐个实现功能特性的方式发布软件，能够获得最好的结果。



进阶阅读：

- 第10章　价值是什么
- 第11章　如何衡量价值

第3章 根据功能特性可以指导得更好



对于任何一个项目，我们知道的第一件事情都是项目的截止日期，至少看起来总是这样。下图用底部标有三角形的竖线来表示该日期。

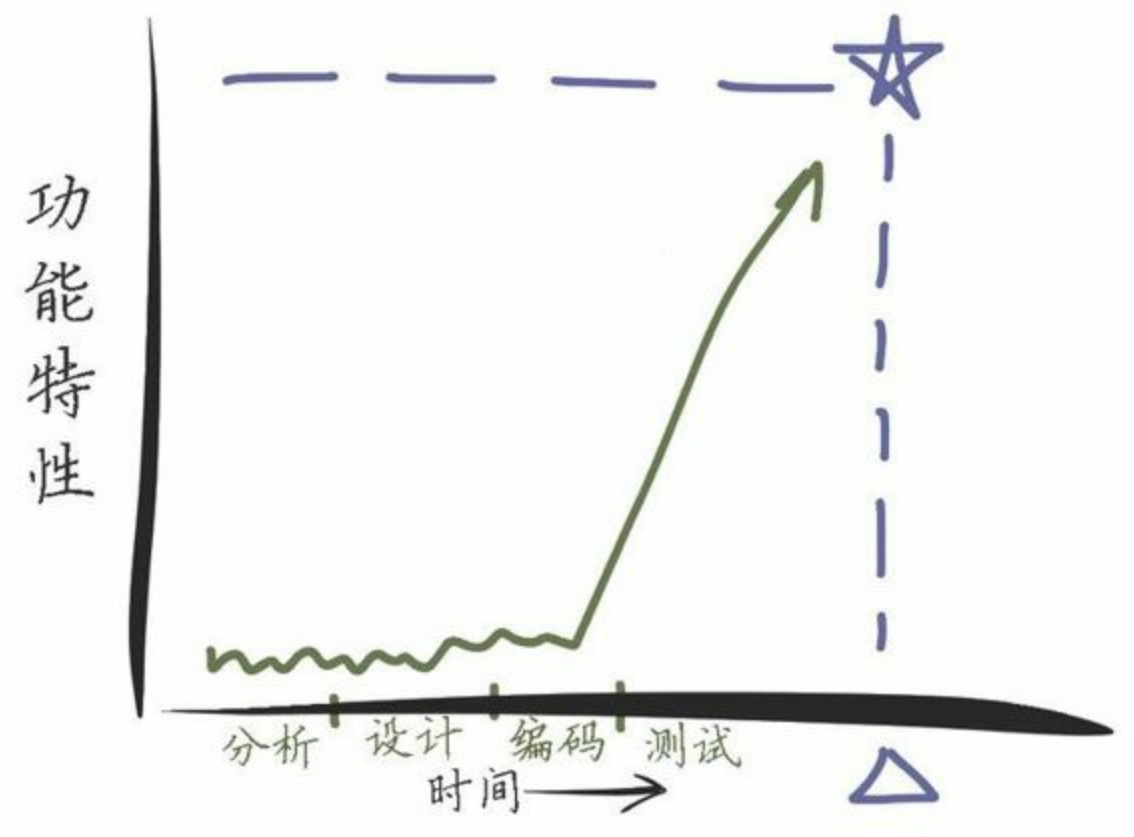
在截止日期到来之前，我们想要实现哪些功能特性呢？毫无疑问，当然想要实现所有的功能特性，也就是图中的横线。五角星则代表我们的计划：在截止日期到来时，我们希望拥有所有的功能特性。这一切都毫无疑问。

然而，由于某些缘故，结果并非我们所希望的那样。通常，最终的结果是：要么少发布了一些功能特性，要么结束的时间比截止日期要晚，甚至是上面两种情况同时发生，也就是下面第二幅图中带问号的线所表示的情形。我们所得到的肯定要少于我们想要的。毕竟，我们想要实现所有的功能特性！

事实是，我们并不能够实现所有的功能特性。我们需要正视这一事实，并进行相应的管理，而不是置之不理；同时需要对项目的运行进行引导，而不是任由它发展。

在你最近的一个项目中，有哪些重要的功能特性没能实现？又有哪些虽然实现了却

被证明是毫无用处的功能特性？你发现得太迟或者几乎太迟的情况有哪些？

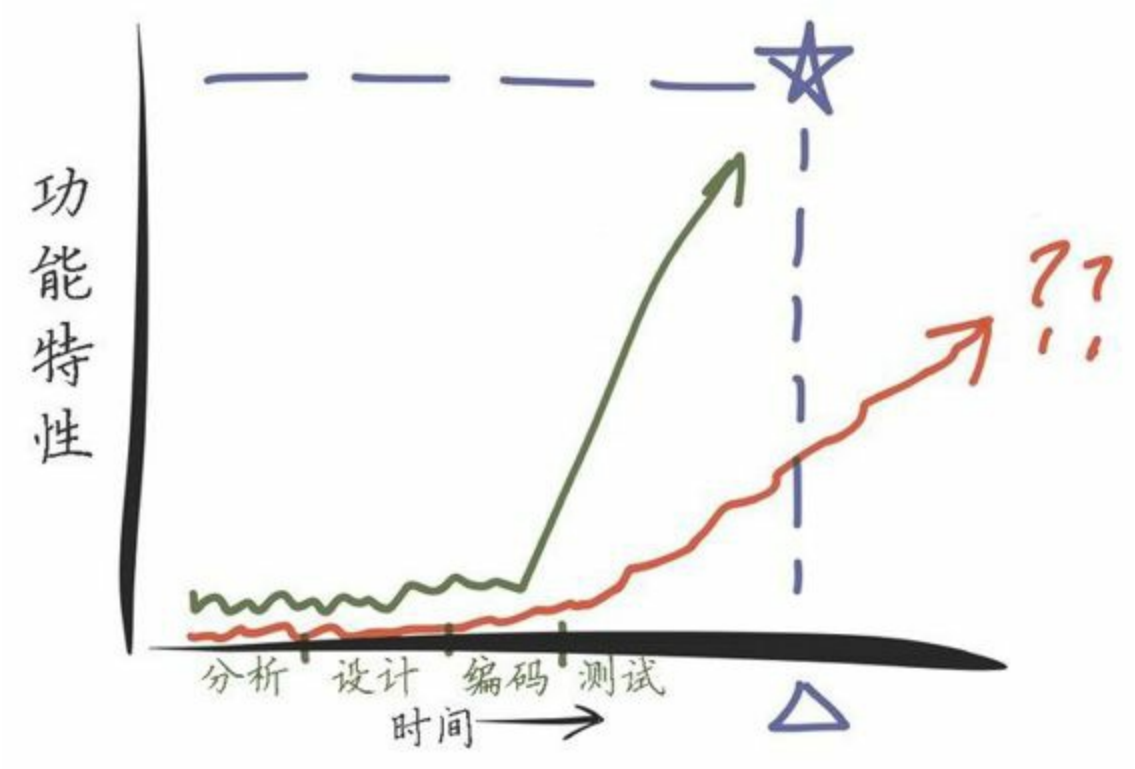


传统的软件项目分阶段进行。

很多软件项目都根据活动来进行阶段规划，项目通常会被划分为四个阶段：分析、设计、编码，最后是测试。上图中带箭头的线表示的就是我们对这样一个项目的计划，看起来十分不错。然而，即使按时完成了分析阶段的活动，我们也并不能据此知道设计或者编码阶段会完成得如何。

直到我们看到软件，才能够真正知道自己做得如何。当我们拿到代码并开始测试时，又会发生什么情况呢？通常，等待我们的并不是什么好的结果！

在你经历的项目中，是否有过当问题产生时你却很少有时间去应对的情况？你认为能够更早地知道实际所发生的情况有价值吗？你曾经是否希望所付出的努力至少能够带来一些价值？



更糟糕的是，事情很少会按照计划发展。

最后，我们开始检查并测试代码，然而现实却并不能让人满意。结果必然是，项目的进度比我们认为的要落后，完成的功能特性也比我们认为的要少。我们已经完成的那部分功能特性并不尽如人意。

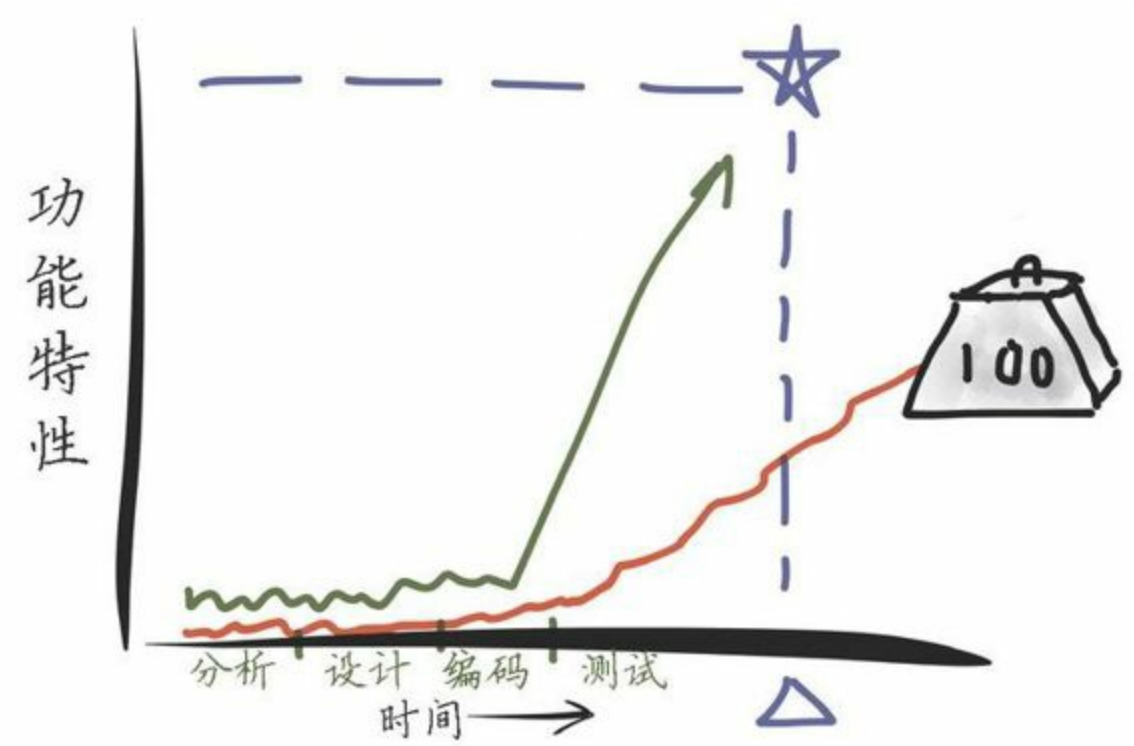
我们知道，自己的要求超出了力所能及的范围，目标设定的本质如此。但当我们发现自己的处境时，为时已晚，这导致我们无能为力。

如果警告信息多一些，或许我们还有希望按时发布部分功能特性。现在，我们只有两种选择：一是结束项目，然而这意味着自我放弃事业；还有就是继续勇敢地前进，希望在自己被放弃之前能够发布一些东西。

无论是哪一种选择，情况看起来都很糟糕。这两种选择都很糟糕！

你是否曾经不得不在很糟糕的情况下发布产品？是否软件在发布时仍有很多缺陷？

是否软件当时已经很难改变？是否缺失重要的功能特性？是否有过想要加入重要的新想法却为时已晚的情况？



基于活动的产品就像是一块巨石。

在像巨石一般庞大的项目中，我们在后期并没有太多办法来节省成本。对于那些永远得不到的功能特性，我们都已经写好了需求；对于那些永远不能完成的功能特性，我们也都进行了设计甚至编写了一部分代码。所有这些工作都只能付诸东流了。

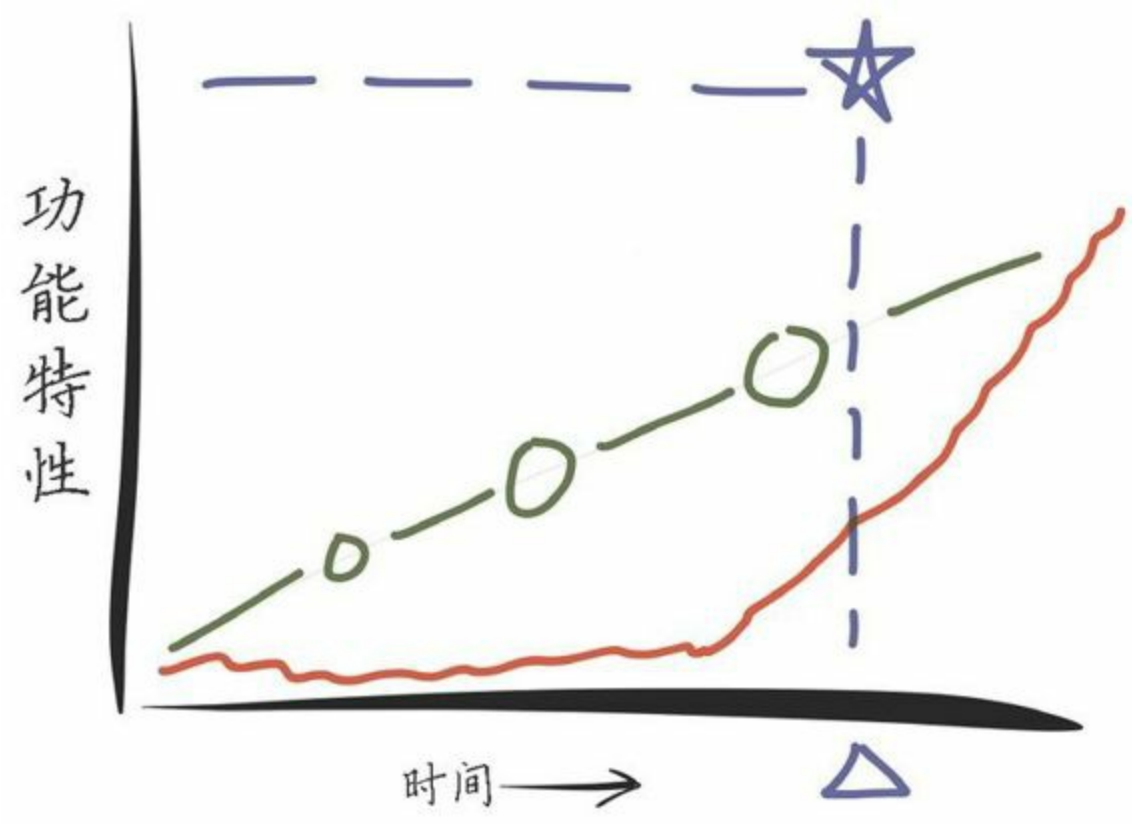
如果了解真实的情况，我们本可以将其中一部分工作推迟。

我们以要么全有、要么全无的心态为这个项目制订了计划：分析并设计了所有的功能特性，还尝试着实现所有的功能特性。最后我们却发现无法完成计划，然而为时已晚。

试图去计划并实现所有的功能特性，这使我们处于不利的境地。我们完全没有时间去改变，而且即使有时间，也不可能理清哪些事情不该做，哪些事情应该做。

相反，我们可以从一开始就计划多次发布版本。多次发版更容易管理，也能够更早交付价值。以这种方式构建软件更容易，同时所有人都能够共赢。

分析、设计、编码、测试，这样的阶段规划真的有助于管理软件项目吗？还是从一开始就关注功能特性，并根据你需要的顺序每隔一段时间就实现一些功能特性这种方法更容易管理呢？下面，我们就来看看这种方法。



根据功能特性交付项目更具有可预见性。

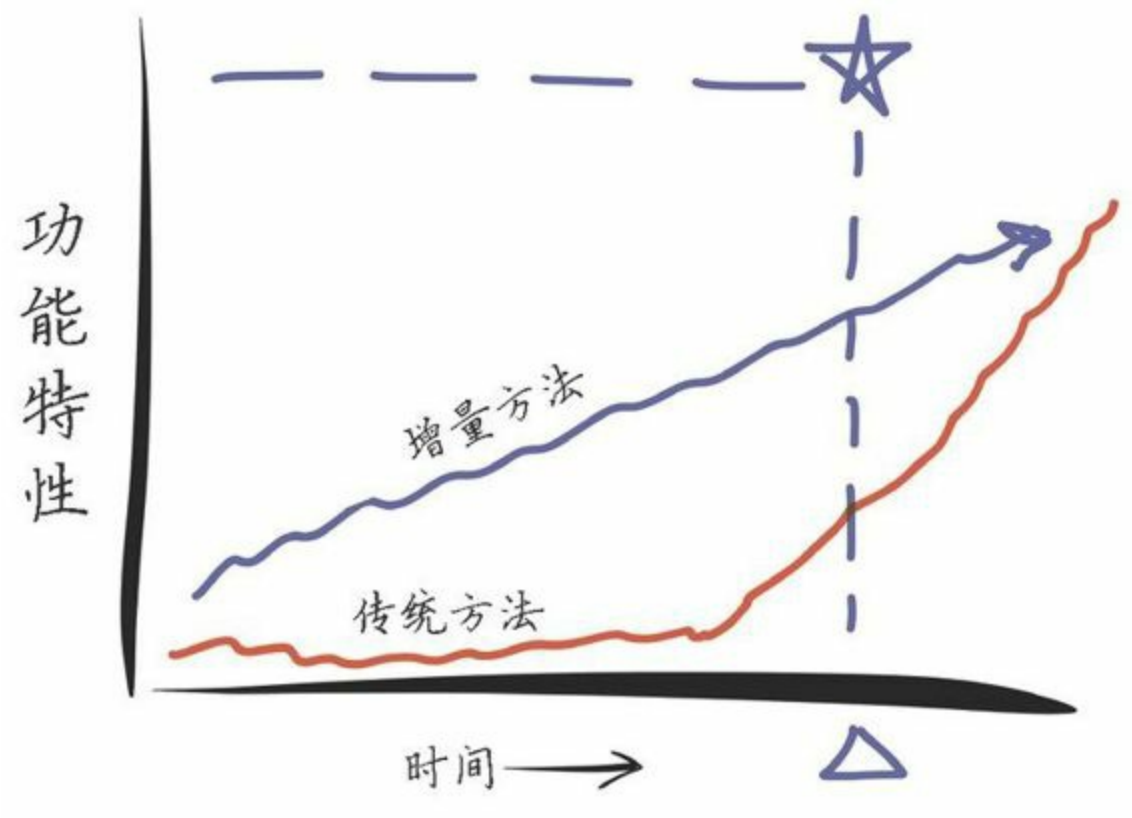
我们已经看到，逐个发布版本和功能特性，可以更早地交付价值。那么我们管理和指导这种实践的能力又怎么样呢？

图中，曲线所代表的传统项目一直不停地在进行着，但传达的信息很少，而且传达

的时间也很晚；而直线所代表的项目却能够频繁、定期地向我们展示真正有价值的功能特性，同时我们也能够看到正在发生的事情，并看到正在形成中的软件！

你能够看出可见的功能特性流怎样更易于管理吗？你知道怎样做可以使项目的价值最大化吗？

风险方面又如何呢？你知道怎样通过构建可见的事物来评估或者降低项目的风险吗？你能够看出怎样用小的试验性功能特性来应对市场的风险吗？



根据功能特性交付项目能够提供更好的信息和指导，同时也会产生更好的结果。

当以逐个实现功能特性的方式构建软件时，情况会好很多。我们能够看到任务完成了多少，也会知道项目进展的速度有多快。这样，我们就能够很好地预测在任意一个给定的日期到来前有多少任务是可以完成的。

我们会选择余下的功能特性中最重要的那个作为下一步的目标。这样，对于任意的

交付日期要求，即使要比一开始要求的日期早，我们也都能够构建出最佳的功能特性组合。在有更好的想法或者用户的需求发生变化时，我们甚至能够变更或者增加新的功能特性。

当项目随着一个接一个的功能特性的完成而逐渐完善时，我们就可以应对真实发生的状况了。同时，我们也能够应对需求以及商业与管理方面的变化。

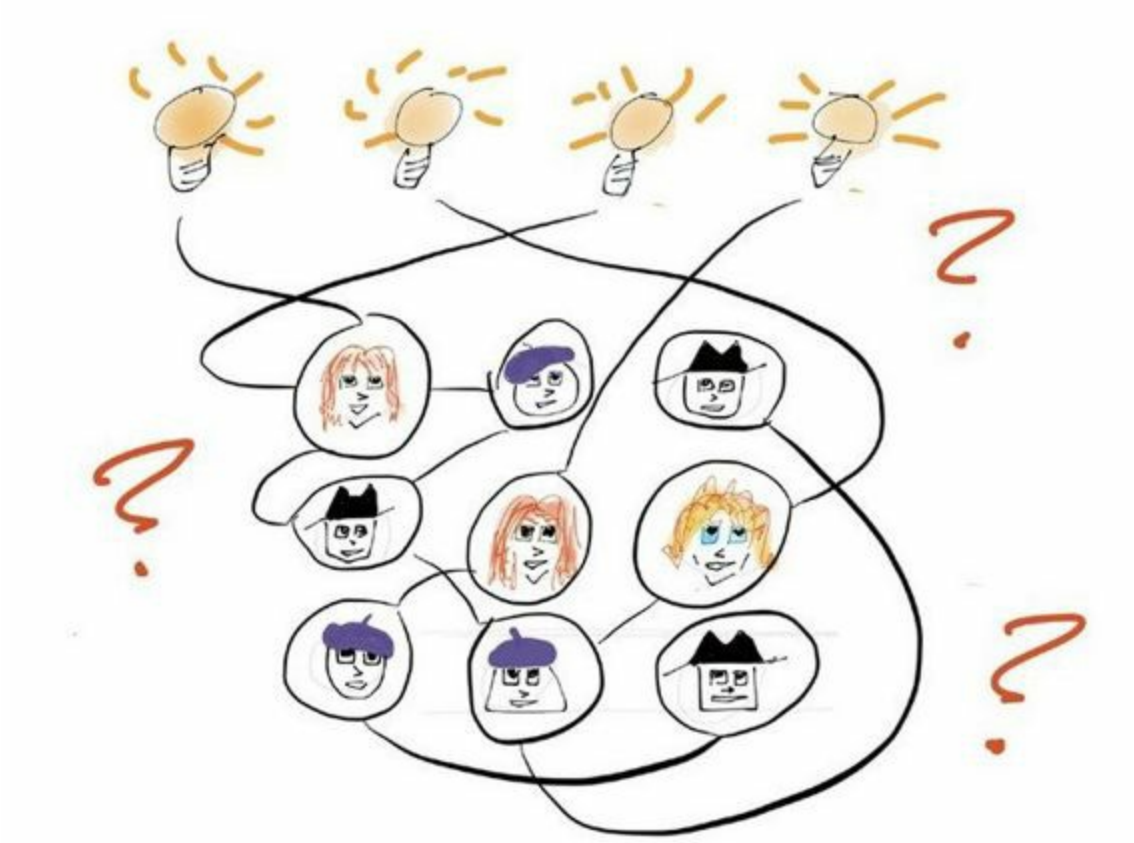
那么，需要具备哪些条件才能使这样的工作方式成为可能呢？在甚至都不知道最终想要什么的时候，我们又怎么能够对项目进行规划呢？

进阶阅读：

- 第14章　组建强大的团队

- 第16章　自然软件开发的管理之道

第4章 根据功能特性组织团队



功能特性的构建需要多种技能

我们想通过功能特性一点一点地获得价值。当通过价值和功能特性来管理项目时，就能获得成功。

那么，要怎样组织工作、组织自己才能最快地获得最大的价值呢？

我们知道，为了完成整个项目，不同的部分需要不同的技能。在具有这些技能的人员全部加入之前，整个项目是不可能完成的，至少不可能完成得很好。

如果根据技能来组织团队，每一个项目都需要经过几个团队的合作交接才能够完

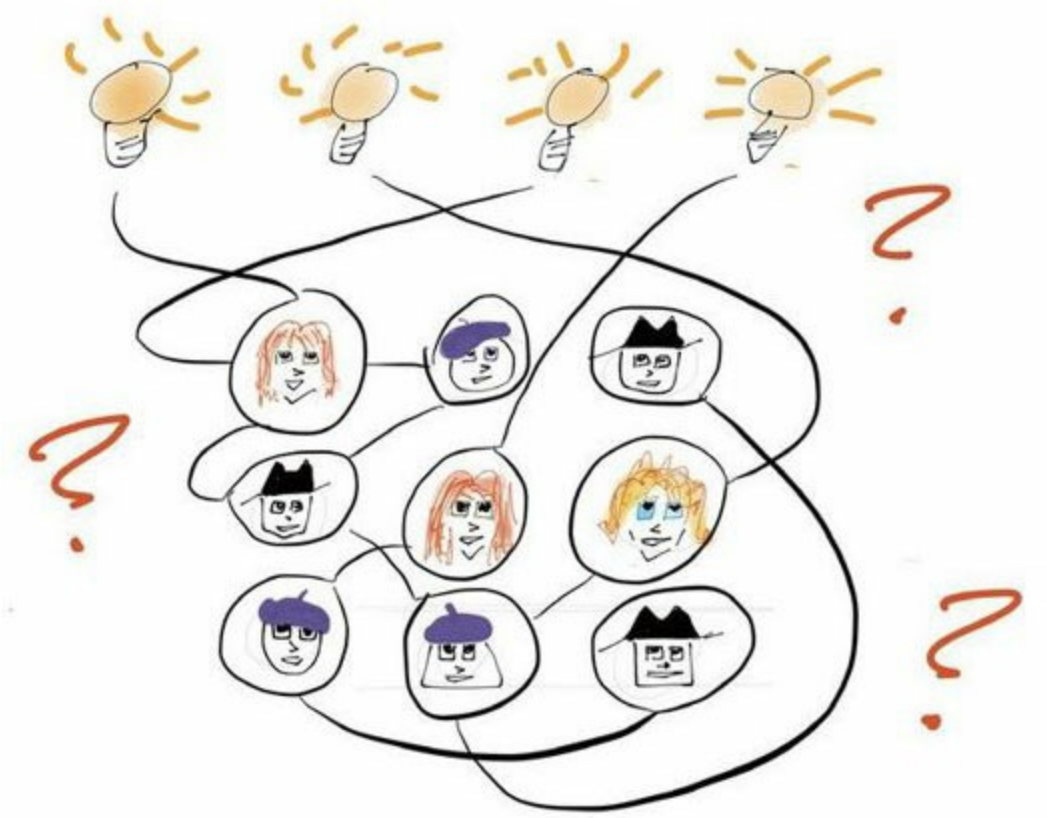
成。每一次交接都需要制订日程表，这会造成进度的延迟。而且极有可能每一次交接都会产生相应的问题。

团队构建功能特性。

这个问题的解决方法很简单：将相关人员组成几个小团队，其中的每一个团队负责构建产品推动人（Product Champion）能够理解的一个功能特性。注意，这里要确保每个团队都拥有构建整个功能特性所需要的所有人员和所有技能。

该方法的优点很明显：我们可以很容易进行工作的跨团队分配，也可以知道工作进展到了哪一步。每一个功能特性都获得了特别的关注，同时，责任和权力是一致的。

这种方法既简单又奏效。但实现起来并不那么容易，不是吗？



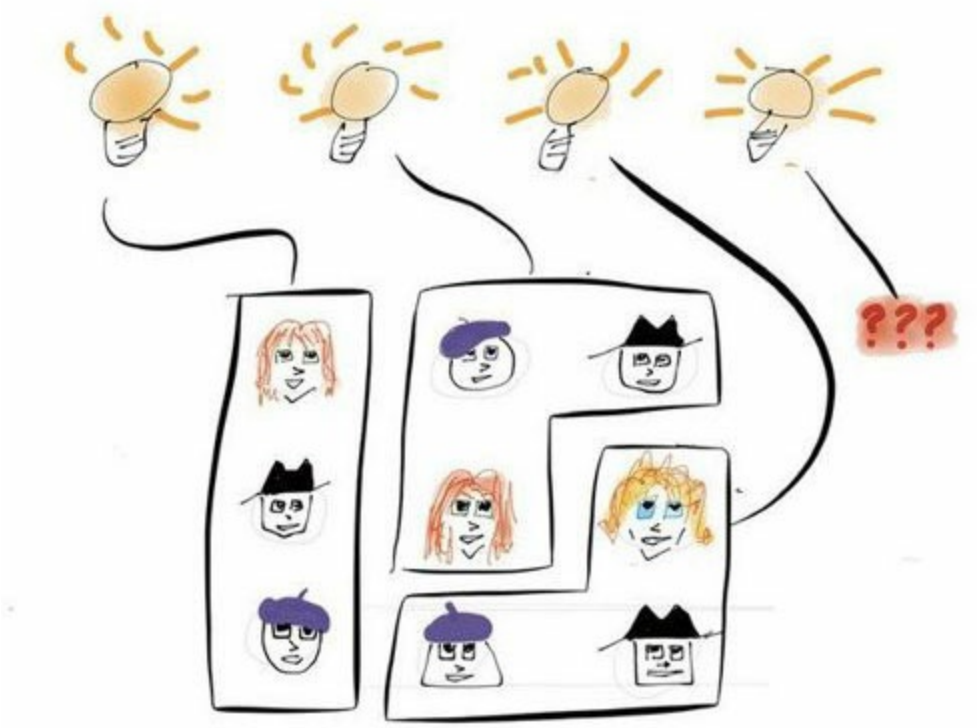
但是……问题是……事实上我们的人员并不是这样组织的。

我知道这一点。而我要说的是，你可能应该这样去做。

如果每一个功能特性都需要经由多个团队的接力传递才能够完成，那么整个项目所需的时间可能会更长，而且最终的质量也可能会更低。为什么？这是因为参与的团队之间或多或少都需要一些协调与配合，而且每一个功能特性都必须由一个团队完成一部分之后再交给下一个团队。每这样交接一次，可能都需要等待一段时间，下一个团队才能够针对该功能特性着手开始工作。而且这一功能特性常常（可以说是大多数时候）都需要再次返回，由第一个团队修改一些之前他们不是很清楚的内容。通常，每一个功能特性都会经历几次这样的来回过程。

这样一来，速度就慢了下来。

是的，功能特性团队对于你来说可能是一种改变。但如果你想要项目能够迅速、顺利地进行，朝着这个方向去转变，很有可能会受益匪浅。慢慢来，别急，尽管去尝试。组建一个相当不错的功能特性团队，看看更少的交接次数是否能够在加快交付速度的同时提高交付的质量。我敢打赌，情况肯定会变得好起来。若果真如此，那就继续这样做下去吧。



我们并没有足够多的专家。

也许你所在的公司并没有足够多的数据库专家或者用户体验设计专家。每一个功能特性团队都拥有一位专家是不可能的。最终的情况可能是，你发现要做的工作远比做这些工作的团队要多。你似乎并不能够组建下一个你所需要的功能特性团队。

好吧，也许情况的确如此。不过，我敢肯定你的公司有一些对数据库相当了解的员工，虽然你并没有将他们称为“专家”。甚至有可能你的一些员工真的具有专家的水平，只是没有专家的头衔而已。很有可能你的很多员工都会设计界面，而且其水平与你最好的用户体验设计专家的水平接近，甚至有可能比他们做得更好。

不妨来做一个思想实验：根据最重要的功能特性优先的原则去组建团队，其结果会怎么样呢？既然你所在的团队正在开发所有待完成功能特性中最重要的那个，那么很明显，它值得你去投入余下的人员中最优秀的数据库技术人员和用户体验设计人员。就将他们安排在团队里吧。其他的以此类推。

很快，你可能会发现：你所组建的这个团队的数据库技术人员或者用户体验设计人员的确水平有限。很好，你为他们创造了一个锻炼和学习的机会。同时，你也创造了组建兴趣小组和实践社区的机会。

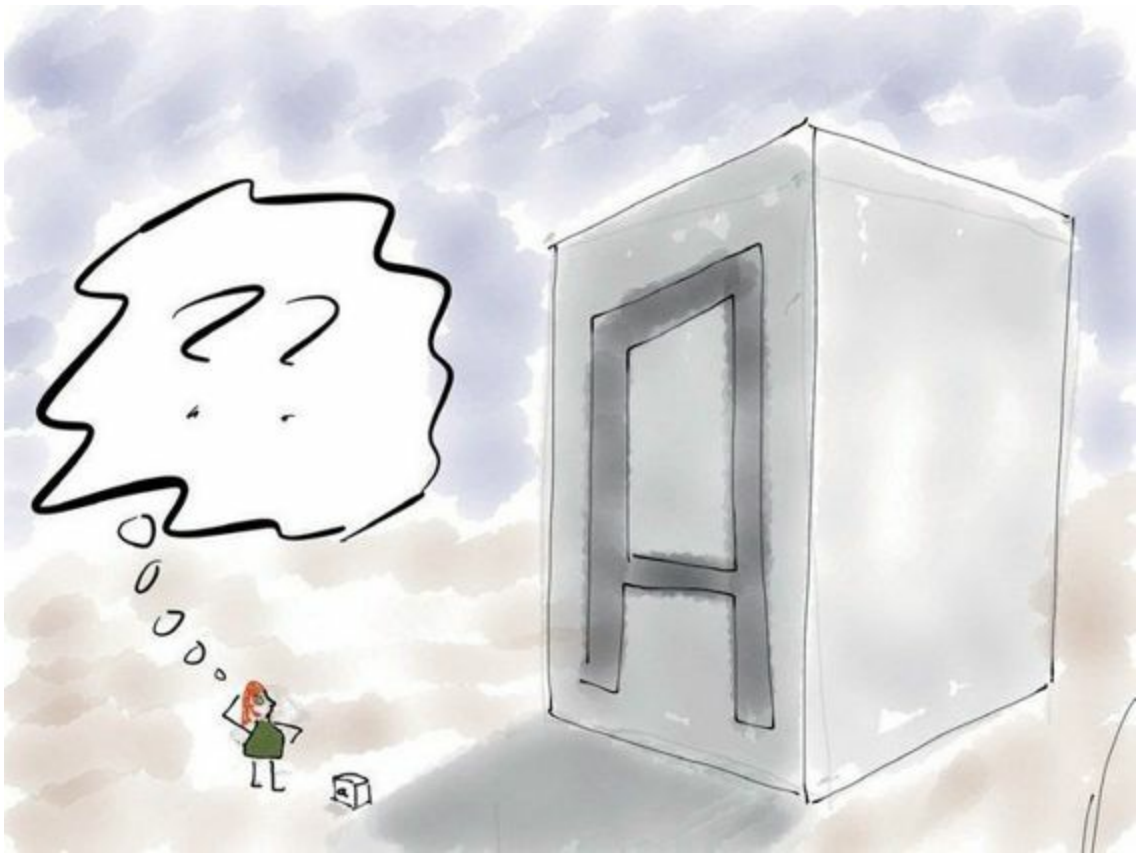
组建实践社区。

现在，让我们来组建一个数据库实践社区或用户体验实践社区。围绕那些曾经在数据库或用户体验组工作过的人员来组建，同时将目前在功能特性团队中负责这些任务的人员纳入其中。这样的社区并不是新的部门：所有人依然属于功能特性团队。

同时，他们也是数据库实践社区或用户体验实践社区的成员，这正如你既是家庭中的一员，同时也可以是高尔夫俱乐部的成员。

你的高级员工（并不总是那些你认为的高级员工）现在有了额外的职责：帮助那些经验不足的人员迅速成长。薪资很高的专家之所以能获得高薪，不应该仅仅因为他是专家，而应该因为他能够帮助其他人成为专家。

你的精锐员工对那些经验不足的人进行了指导。他们融入了其他的团队并去帮助这些团队的成员，这样能够保证这些团队的成员在完成自身职责的同时，了解需要做什么以及怎么做。很快，你就会拥有所有你需要的专家，当然还有更快乐的员工！



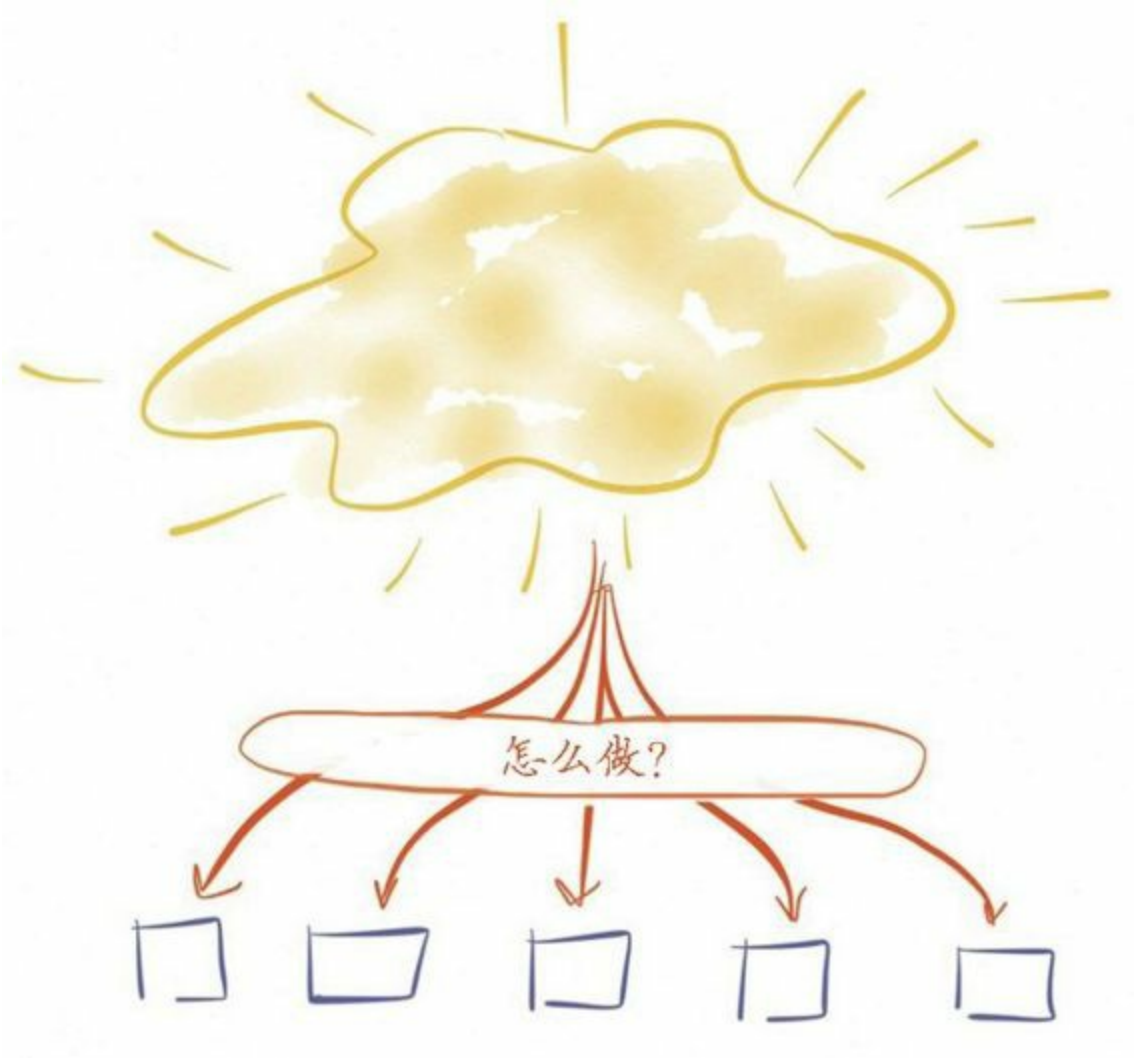
功能特性团队使得“规模扩展”很容易。

本书的第二部分有一章专门讨论“大规模”的问题。简单地说，大部分工作都可以由一个跨职能的团队完成；剩下的大部分工作则可以由几个功能特性团队并行进行，以实现那些你理解并需要的功能特性。

你的公司或许很少真正需要几个团队步调一致地工作。如果将任务按照功能特性来划分，你很有可能会发现剩下的事情都很简单。这可以使你无需花钱“扩展”公司，从而节省一大笔开支。

一句话，根据功能特性组织团队。你将为此决定而感到庆幸。

第5章 根据功能特性进行计划



从愿景到细节

当我们经常性地发布软件版本时，情况会最好：价值会增长得更快、更好；管理层每隔一段时间就能够看到项目的进展；同时，由于目标小而明确，开发工作也能够以最好的方式进行。

然而，产品的“愿景”总是从伟大的想法开始，虽然朦胧却很诱人。愿景是关于产品的大的思路，而不是小的功能特性。

那么，怎样才能够将宏伟、美好的产品愿景落实到详细的功能特性上，从而使我们获得最好的可视性与控制能力呢？

做计划是必要的。

艾森豪威尔将军曾说过：“计划本身是无用的，但做计划是必要的。”我们的确需要针对产品认真思考，不只是一开始，而是一直需要这样做。

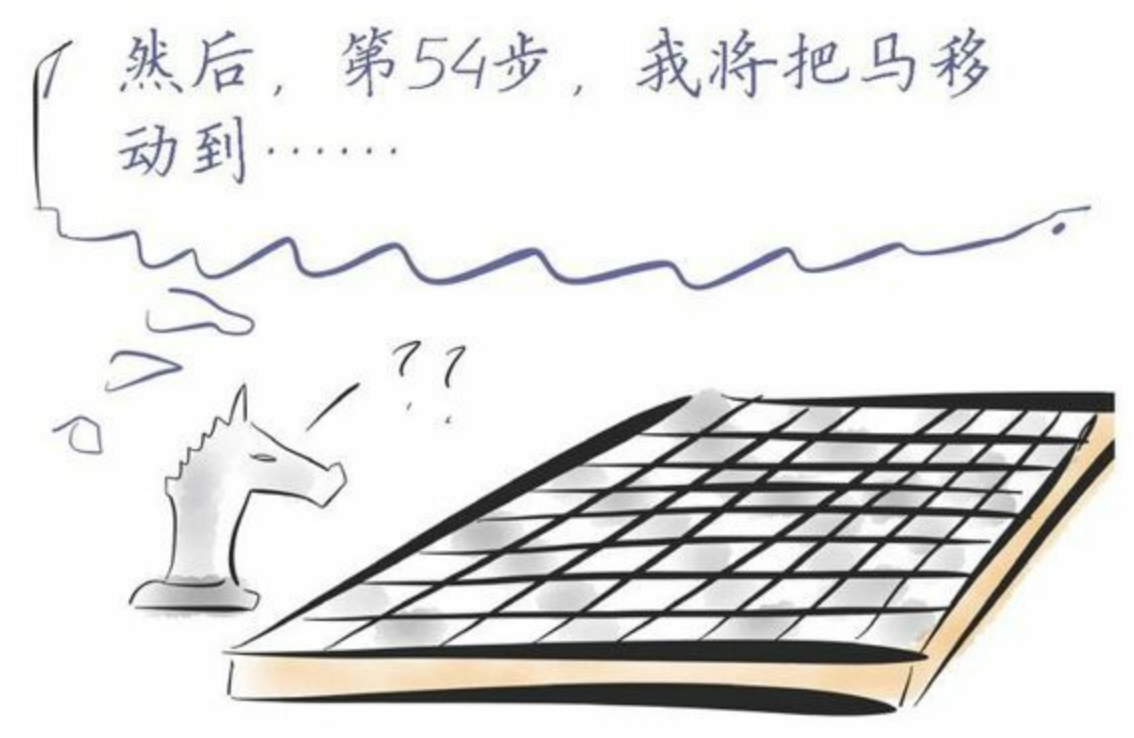
的确需要做计划，但并不需要详细列出一张什么时候发生什么事情的清单。我们可以在事情发生时，再去决定下一步该怎么做。太过详细的计划只会浪费时间，并造成困惑。

尽早确定哪些核心的功能特性必须尽快有，哪些功能特性不能没有，这很重要。我们要确定这两种功能特性，并将它们记录下来。

同时，对于那些价值很小的想法，我们需要无限期地延迟实现。将时间花费在思考这些功能特性并对它们进行跟踪分析上，实属浪费。

然而，做计划是很重要的。很可能在仔细考虑了很多不好的想法后，我们才能够得到几个不错的想法。因此，我们需要做计划，同时还要保持轻松，并做好迎接变化的准备。

不妨对以往那些有很多功能特性的项目进行反思。其中有多少想法完全实现了？又有多少想法真的很不错？是否其中有些想法根本毫无用处？我过去的很多想法正是如此！



详细的计划是无用的。

如果做计划是不错的主意，那么更大的计划是否更好呢？我们的头脑中有一些很模糊的想法，这些想法可能会被大家认同，也可能不被认同。但是由于某种原因，我们觉得有必要估算出实现每个想法所需要的时间。这样就可以将所有这些想法加在一起，确定从周二开始算起一百天后哪些想法能够完成。

关于有多少软件项目的最终支出远远超出其预算，我们都读到过可怕的统计数据。

或许软件开发人员不善于估算预算，因此他们需要更加努力。没错，所有人在估算方面的能力都欠佳。我们不只需要更加努力，还应该找到更好的方法。

更好的方法如下：一是确定项目的时间期限和开支预算；二是优先开发那些最优价值的功能特性；三是确保产品能够随时发布，并在时间结束时立即停止。开发工作甚至很有可能会在截止日期之前停止，因为我们已经得到了最重要的功能特性。这样就能够在更短的时间内，以更少的花费交付大量价值。

为了给项目设定预算，需要知道哪些长期的细节？为每件事都做计划并试着将其纳入总的预算，或者确定比较紧的时间期限并在此期间尽可能构建出最好的功能特性，哪种做法更好？

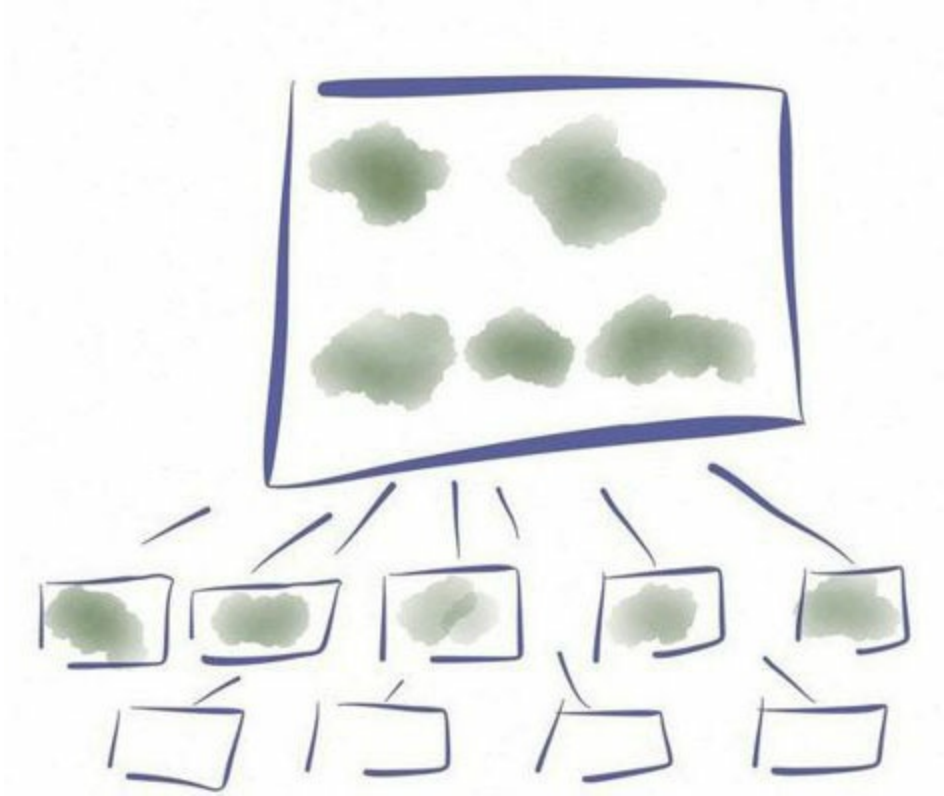
准备开始

理想情况下，项目可以从一开始就启动。先有一个想法，再稍作考虑，然后组建一个小团队，就可以开启整个项目的构建之旅了。运用这种方法，我们很快就能够知道是否可以创造出有价值的东西，以及整个项目大约需要多长时间。然后，我们就能够决定是削减投资还是保持投资不变，抑或是追加投资。

然而，公司的运作方式有时并非如此。早在做出投资决定之前，投资人就坚持要知道拟议的项目到底需要多长时间完成，是需要几周还是几个月，抑或是几年。或许，我们可以要求组建一个团队并运行这个项目一段时间，从而给出这一问题的答案。要是我们能够做到的话，就可以按照这个方法去做，并且可以快速学到很多。不过，尽管这种方法很合理，但是有时候它并不可行。我们需要一种方法对整个项目的大小进行初始的划分。

关于如何进行估算，人们已经进行了很多讨论。然而，依然有很多项目不是发布的功能特性太少，就是发布时间很迟。如果你真的需要找到更好的估算方法，我建议不妨尝试下面这种方法：确定项目的总体预算与截止日期，找到一位产品推动人来决定功能特性的开发顺序，并组建能够随时交付软件的团队。

那么，对于拟议的项目，“他们”需要知道多少？当我们针对该项目出价时，多接近才算估算准确？我们能凭着粗略的估算获得成功吗？



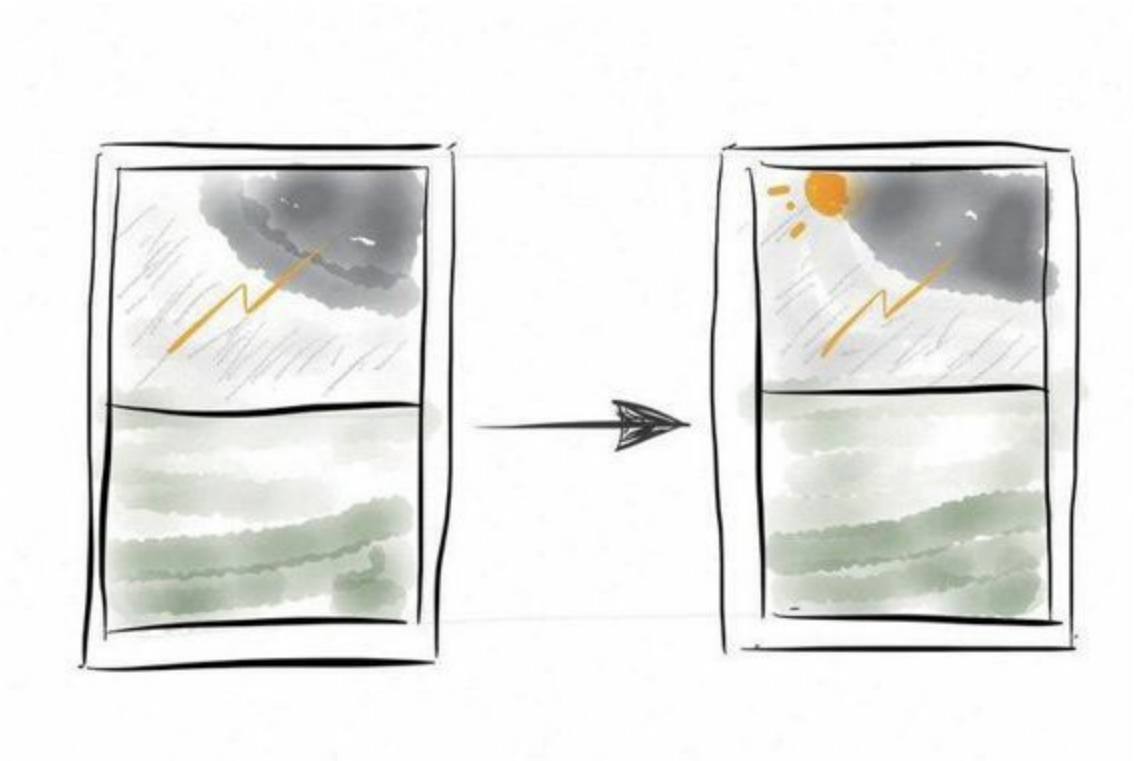
持续计划：分解功能特性。

只在项目的起始阶段做计划还远远不够。因为我们关注的是价值，所以需要一直做计划。团队应该以固定的节奏工作，这样的工作周期通常被称为迭代（iteration）或冲刺（sprint），其长度一般为几周。每个功能特性（通常被称为故事）最好都只需要两天或者三天的时间就能够完成。

我并不推荐将故事划分得很大，然后再将其拆分成技术性条目（通常被称为任务）。如果采用任务的方式进行迭代，一方面会导致业务人员不能清楚地了解项目的进展程度，另一方面也会使得他们在为期两周的迭代期间不知道如何提供帮助。因此，我建议坚持采用故事的方式，这样工作才能更好地进行。

当然，将大的故事划分为更小的、业务人员能够理解的故事会更好。在我看来，这样做总是可能的。一开始会比较棘手，但经过几小时的练习后，团队很快就能够学会怎样将功能特性划分得更小，而不是将其分解为技术步骤。

针对你的产品，思考它必须有的某个大的功能特性。怎样将它划分为更小的功能特性？其中的某些功能特性是否比其他的更有价值？这说明了什么？



团队到底需要承担多少工作？

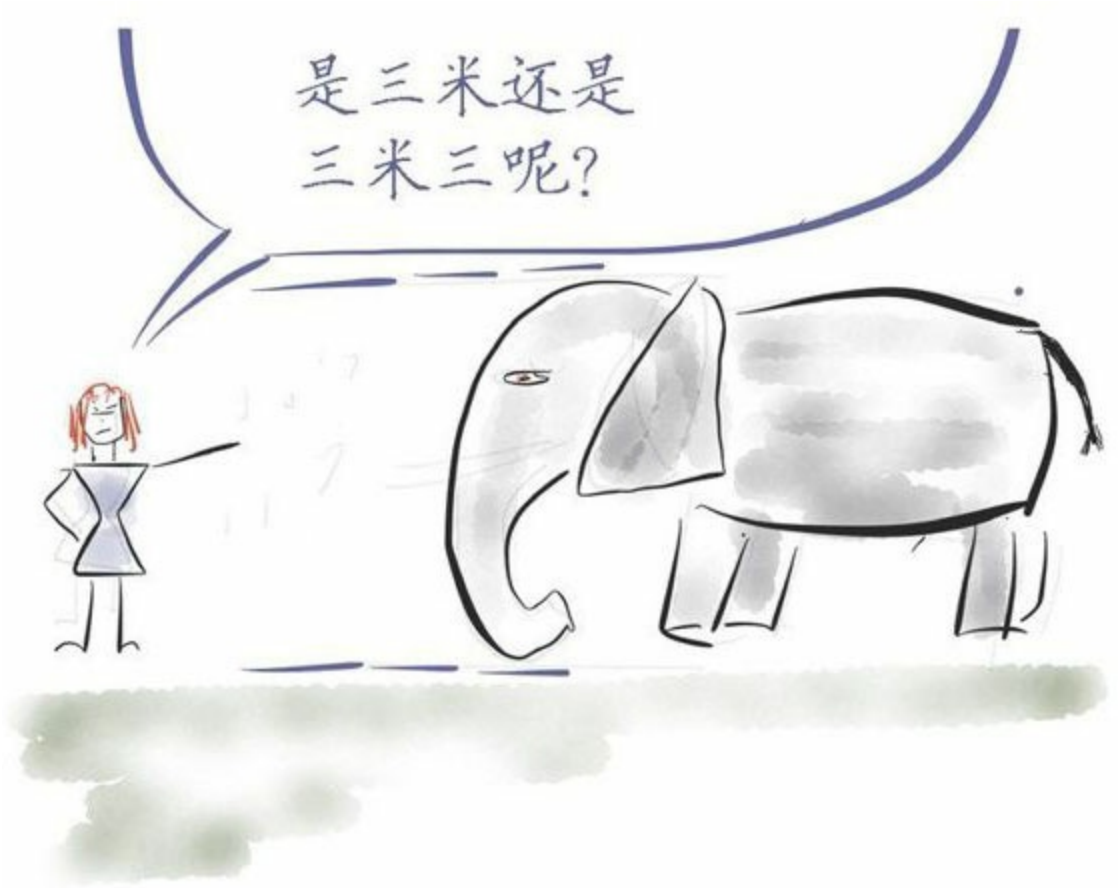
这个问题的答案很明确：应该由团队自己来决定在接下来的两周里到底能完成多少工作。在该问题上，他们最有发言权，而且如果决定是由团队自己做出的，他们会更加投入。有很多方法帮助团队决定完成多少任务，所有这些方法都基于“昨日天气”（Yesterday's Weather）这一想法。我最初是从Kent Beck和Martin Fowler那里学到这个想法的。

你今天能够完成的工作量很有可能与昨天完成的相同。在按迭代进行的项目中，根据上一个迭代完成的工作量来计划本次迭代，使两次迭代完成的工作量大致相等。

每个迭代开始前，我们就需要做好计划。为了确定每个迭代能够承担多少工作，需要理解要进行的工作。对此，团队成员需要一起讨论。产品推动人逐一展示每一个

功能特性，接下来整个团队简要讨论完成这一功能特性都需要做哪些事情。团队的每个成员都需要参与其中，使整个团队在着手工作之前理解所讨论的功能特性。

我并不推荐针对单个工作任务进行估算。相反，我们需要先理解它们，然后看看总体需要多长时间，最后再决定整个团队能够完成多少工作。如果估算确实能够对团队有所帮助，那就继续这样做。不过应该小心！重要的并不是进行准确的估算，而是能够以一贯的节奏做好工作。



估算有风险！

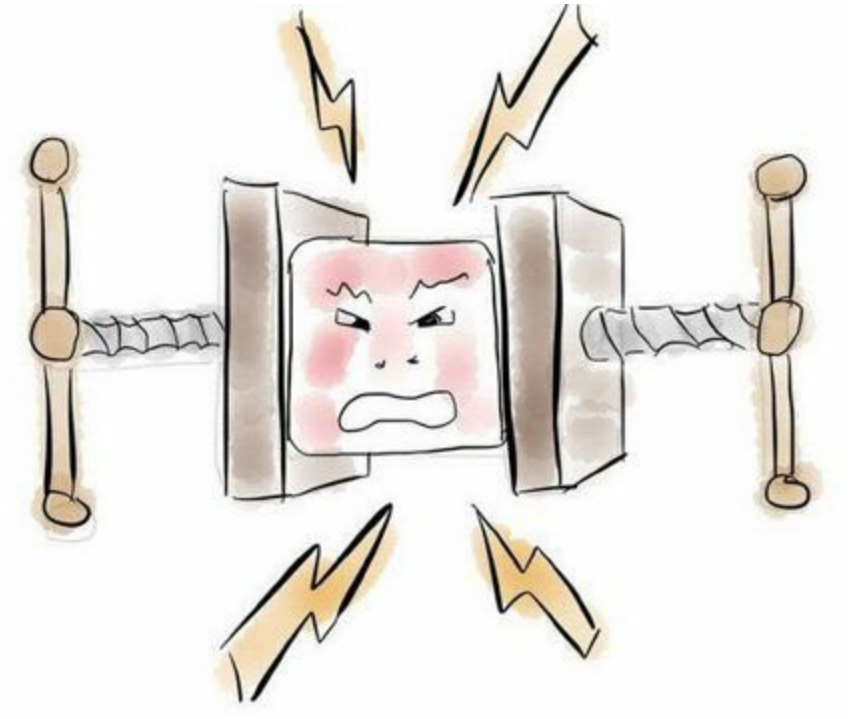
估算有一些极大的风险：我们几乎是不可抗拒地想要对估算进行“改善”，或者是进行比较，而这两种行为都十分有害。需要谨记的是，无论从商业角度还是管理角度来看，要获得最好的结果，需要明确各项工作是按时完成还是推迟进行。而将精力集中在估算上则会影响工作的完成，同时由于希望估算得比较准确，所得出的估

算结果几乎可以肯定是保守的。

有不少团队根本就不进行详细的估算，照样运作得很好。他们先思考要进行的工作，再将其拆分，通常是一直拆分到具有独立测试价值的故事，然后就开始工作。当需要估计任务完成时间时，他们只需要算一算完成了多少任务即可。

诚实地面对自己，你是更愿意知道估算的情况，还是实际的情况？更准确的估算能够为你带来什么？哪一个可能会妨碍项目的范围管理？除了改进估算之外，还有哪些事情能够通过管理的方式更好地完成？

如果更准确的估算就意味着更保守的话，你还宁愿要它吗？难道准确的预测要比掌控实际情况更好？



根据“挑战性的目标”制订计划，危害性很大。

在制订计划（特别是短期计划）时，人们往往会设定“挑战性的目标”，或者“鼓励”团队“再多完成一个功能特性”。请不要这样做，因为这会带来很严重的后

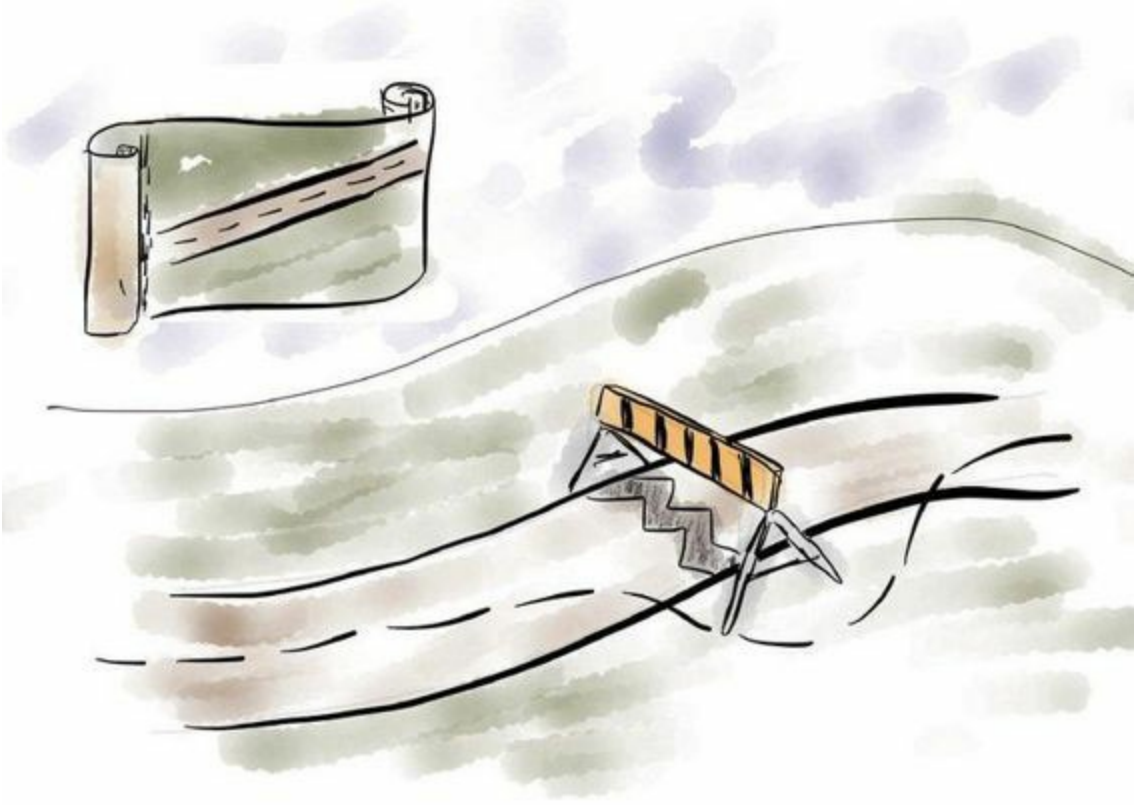
果。原因是，团队真的会尝试这样做。由于急于讨好上司，他们会不自觉地赶工。为了能够在计划的时间内多完成一个功能特性，他们会遗漏一些测试，并且所写出的代码也不像以往那样清晰整洁。

赶工将会给整个项目带来更多的缺陷。与防止缺陷发生相比，修复缺陷则需要花费更长的时间，因此赶工不但不会加快项目的进度，反而会影响进度。更糟糕的是，当项目接近尾声，越是在你不希望进度延迟的时候，它越会影响进度。

糟糕的代码同样也会影响进度。如果代码清晰整洁，下一个功能特性也会进展得更顺利；反之，如果代码很糟糕，所有的功能特性都需要花费更长的时间。

压力具有破坏性，请尽量避免向团队施压。

在你所经历的项目中，压力造成了哪些不好的影响？它是否造成了缺陷数量的增加？是否花费了你更多的时间？是否消耗了更多宝贵的人力？什么时候压力会真正有所帮助？除了压力，你能够想出其他同样可以获得这些好处的方法吗？

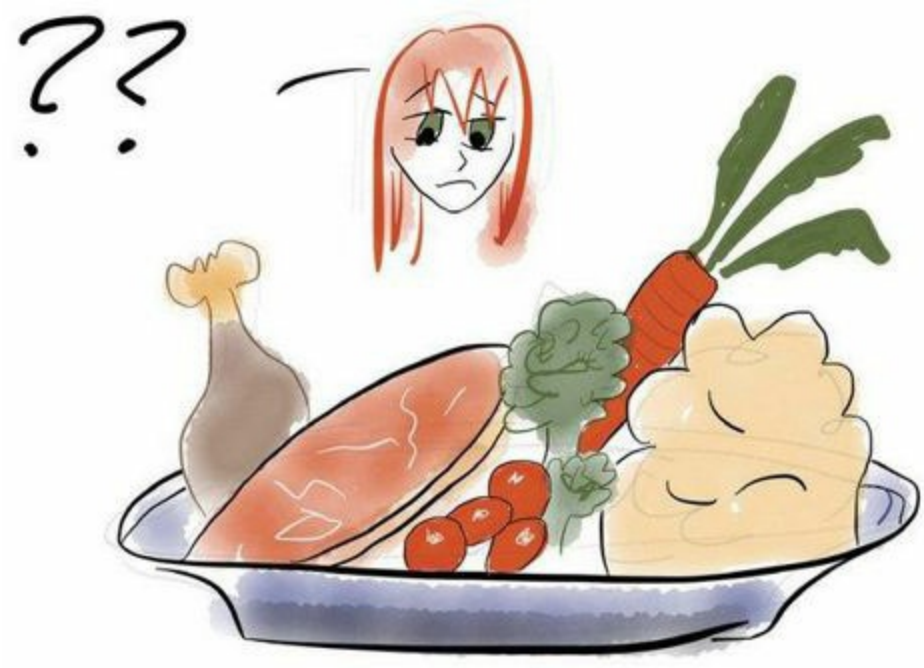


在不进行估算的情况下工作。

一旦能够熟练地将所有的功能特性都拆分得差不多大，我们就能够很好地管理项目，因为我们已经清楚地知道项目需要多长时间完成。由于我们的工作就是安排任务完成的先后顺序并优先完成那些最有价值的任务，因此在没有估算压力的情况下，我们可以引导项目走向成功。

时至今日，估算仍然是软件开发中一个颇有争议的话题。许多人认为估算与详细计划很重要，也的确有一些公司很想这样做，仅仅是这一点就给了我们进行估算的足够理由。糟糕的是，对于估算，我们几乎总是做得很差。对于正在进行开发任务的团队来说，估算最多只是内部事务，很有可能完全是浪费时间。

一般说来，估算结果很可能会是错误的，而且它会使我们将注意力集中在成本上，而忽略价值。不再强调成本的估算或是减少这方面的估算，将注意力放在价值上，以此实现成功。



常做计划，确定下一步要做的事情，不要吃太多。

随着项目的进行，每隔几周就做一次计划，通过计划来决定接下来要做的最重要的事情，同时团队也会明确他们要承担的任务。首先选择那些最有价值的想法——这也是使价值以最快的速度增长的方法。

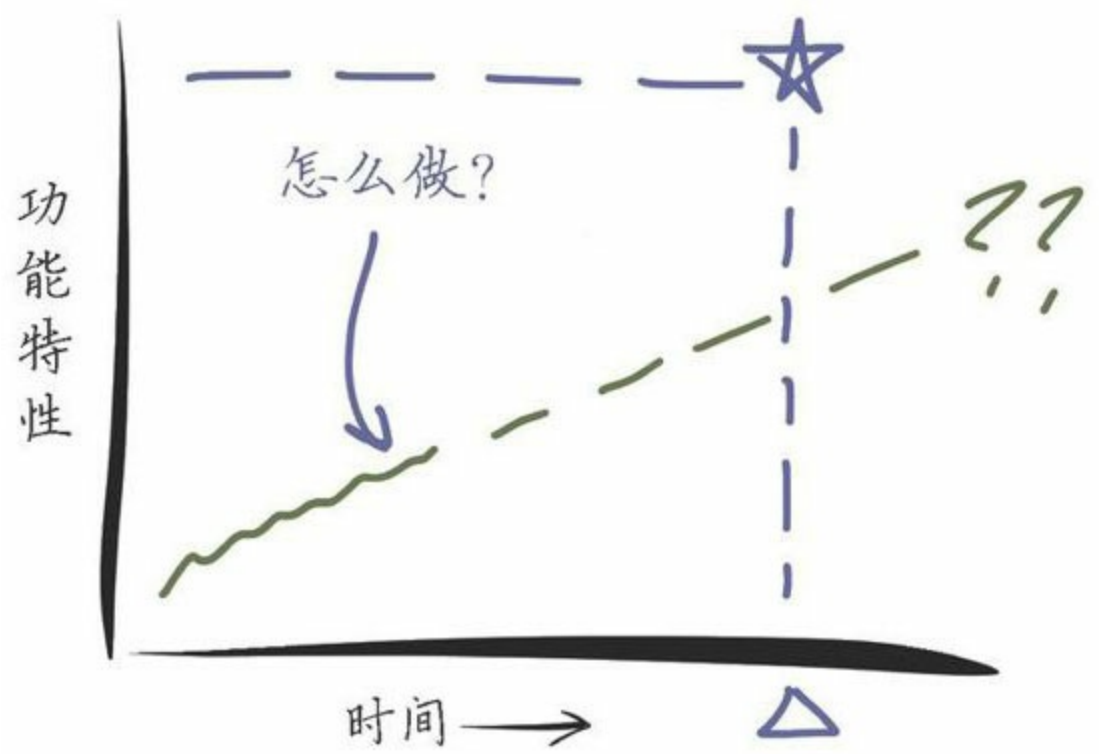
即使是在典型的为期两周的迭代期间，我们仍然期望能够学到一些东西。有时我们会太过保守，在迭代结束前还能够完成更多的任务；更多的时候，我们过于乐观，或者是由于压力而给自己安排了过多的任务。

当你的餐盘里有太多的食物时，请不要全部吃掉。那样你会变得肥胖，而且容易犯困。臃肿、让人厌烦的代码是不利于工作的。与其同时做几件事而每件事都做不好，还不如专心致志地做好一件事。一旦认识到团队承担的任务太多，就需要从任务池中移除一些。之所以需要这样做，是因为保持良好的状态对于完成整个项目来说十分重要。

进阶阅读：

- 第15章　使用五卡法进行初步的预测

第6章 根据功能特性构建产品



根据功能特性来构建产品，能够使我们交付更多的价值。由于开发团队能够演示软件，因此无论是管理还是做计划都很容易。然而，这种方法真的可行吗？开发人员真的可以根据功能特性来构建软件吗？

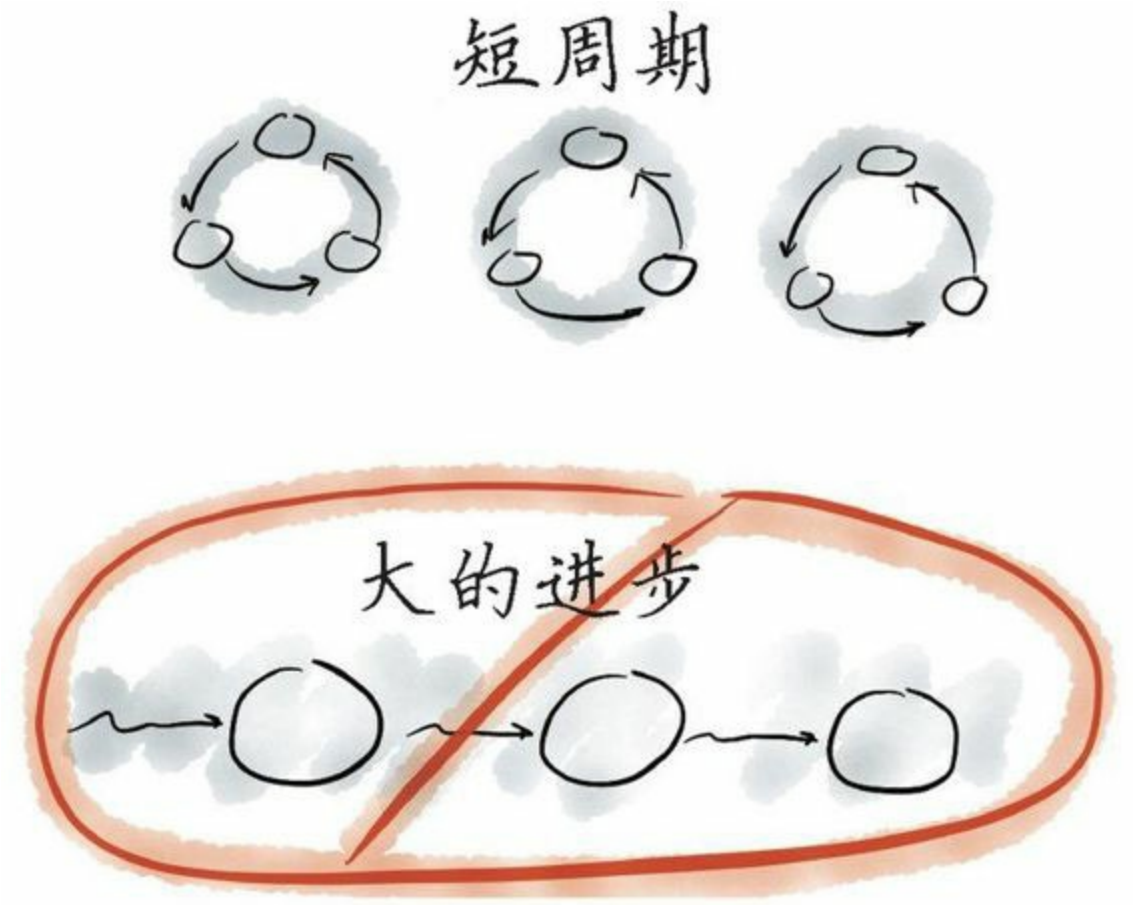
答案是：当然可以！根据功能特性开发是十分可行的方法，几十年来有很多团队应用此方法并且取得了成功。此外，还有上百家企业和机构学会了如何应用它。人们正在用这种方法构建所有你能够想到的软件类型——当然，这也包括你正在构建的那种软件。

你以及你所在的公司都会因为采用这种方法而受益。每个人都需要进行一些学习，并做出一些改变。由此所带来的管理的提升与回报的加速，完全值得我们付出努

力。

下面，我们首先快速地看一下要采用这种方法，都有哪些要求；然后再深入探讨一些细节。

如果想要频繁地交付小的功能特性，我们通常需要做些什么工作？这种方法有哪些缺点与不足？



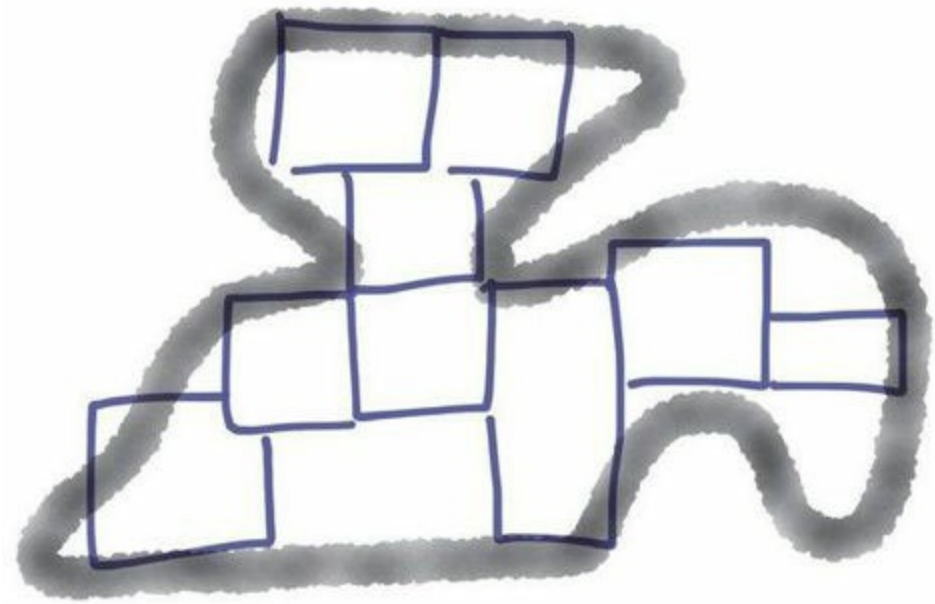
在每一个短周期内，完整地构建一个小的产品。

在计划和管理项目时，我们将其划分为多个为期一至两周的短周期。在每个周期内，确定需要完成哪些功能特性，并清楚地说明如何测试它们。这样，开发团队就可以根据这些要求构建功能特性，由此我们也可以验证功能特性是否通过了测试。

在每一个为期一至两周的迭代中，将完整的产品开发流程走一遍，从最初的概念到准备发布的产品。也许一开始我们做得并不好，但在经过几个迭代之后，就能逐渐适应整个过程。适应了之后，预测整个项目能够划分为哪些连续的功能子集对我们来说会变得越来越容易，而且我们头脑中最终产品的样子会一次比一次清晰，同时我们也更清楚怎样去做。

每一次迭代的过程也是学习的过程。我们会知道在一至两周的迭代期内能够完成多少工作量，会学习如何检验是否完成了所要完成的任务，还会学习在不增加费用的情况下定义功能特性的最有效的方法。同时，我们也会学习怎样在保持编码和设计清楚且灵活的情况下一步一步地构建功能特性。

如果开始采用这种短周期迭代的方法开发完整版本的产品，你预计会遇到哪些困难？可能会出现什么问题？你必须学习什么？



细化产品愿景。

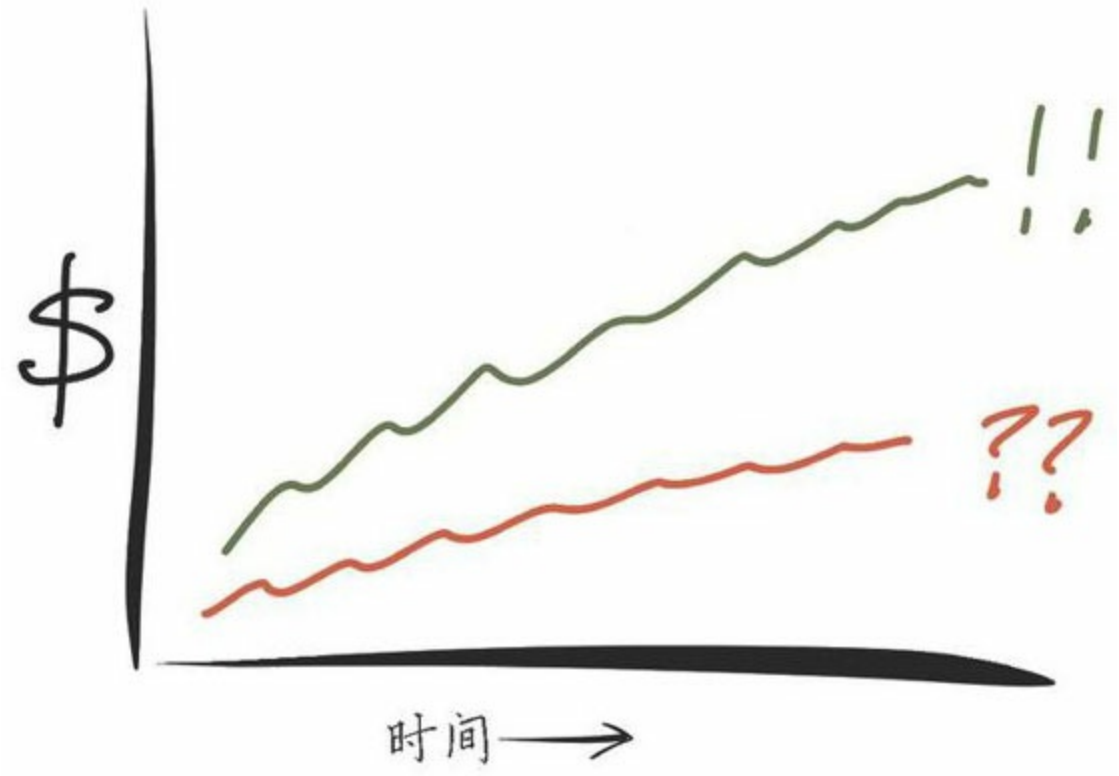
在根据功能特性构建产品时，每隔几周我们都会要求开发团队构建出新的功能特性。我们希望看见所构建出的是我们真正想要的，而不是一些我们不懂的技术或者

类似的东西。

这就意味着业务人员必须要有能力将大的、模糊的、笼统的需求切分为小的、切实可行的开发步骤，这样才能够用尽量小的努力去获取最大的价值。要想做好这件事，业务人员需要得到整个公司的帮助与支持，甚至请求外部支援。通常来说，这需要整个业务团队下很大的功夫，而我们则需要相信业务团队能够弄清楚他们需要什么。

这并不是简单的任务，而且对于成功至关重要。我们必须要明确哪些功能特性对于最终的产品来说是必不可少的，又有哪些属于“锦上添花”的性质。搞清楚了这些，我们就能够从软件投资中更快地得到回报。

你的产品都有哪些大的功能特性？哪些小模块能够帮助你看清楚目前你做得怎么样以及下一步要做什么？现在，我们应该向你展示什么样的软件？

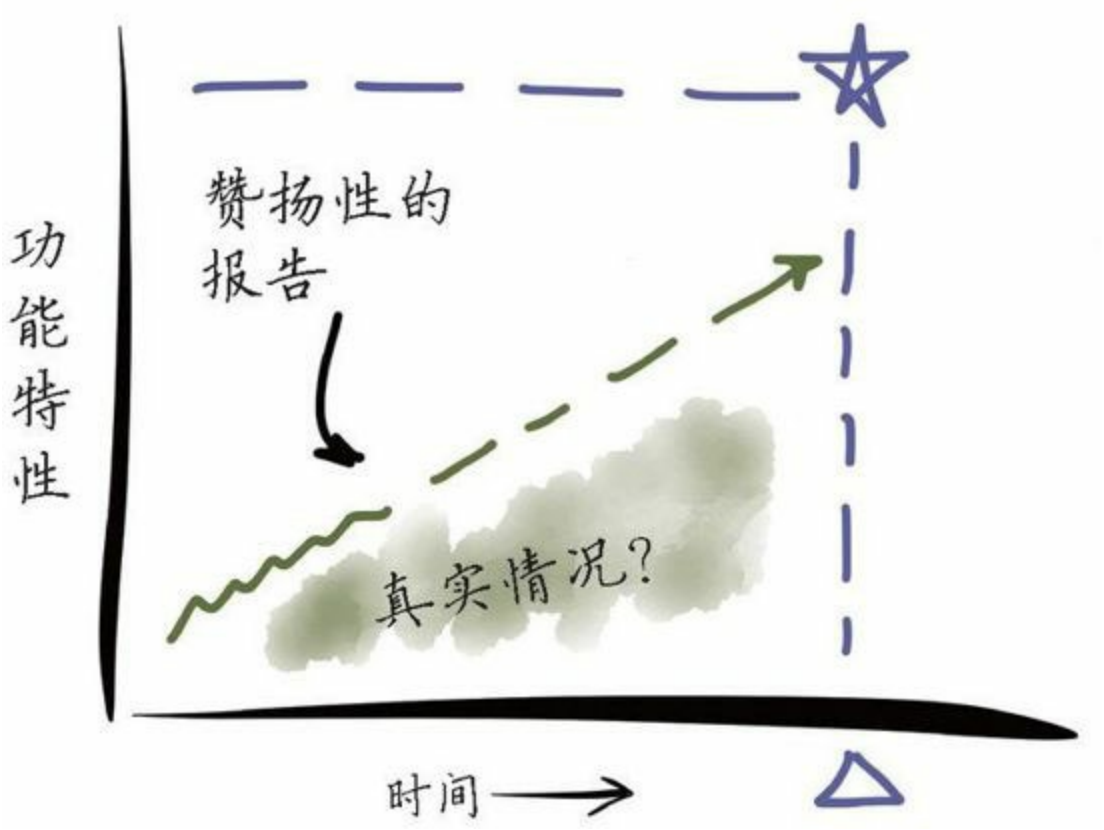


总是将价值可能最大的任务列为下一个目标。

随着经历的迭代越来越多，我们会越来越清楚迭代结束时能够完成多少任务量。优先完成那些最有价值的功能特性，这一点至关重要。整个团队一起合作，共同确定有能力完成且价值高、成本低的功能特性。这样一来，团队就学会了怎样在有限的时间和预算内构建最好的产品。

我们需要公开地去做所有这些事情。团队的每个成员都需要看到实际的进展情况。我们不接受诸如“完成了90%”这样的表述。功能特性要么“已完成”，要么“未完成”，不存在中间地带。必须清楚地看到项目内部真实的进展情况，这样才能够将项目引向最好的结果。

当一个功能特性“完成”时，我们需要知道哪些情况？什么会影响我们指导项目和提升产品的能力？



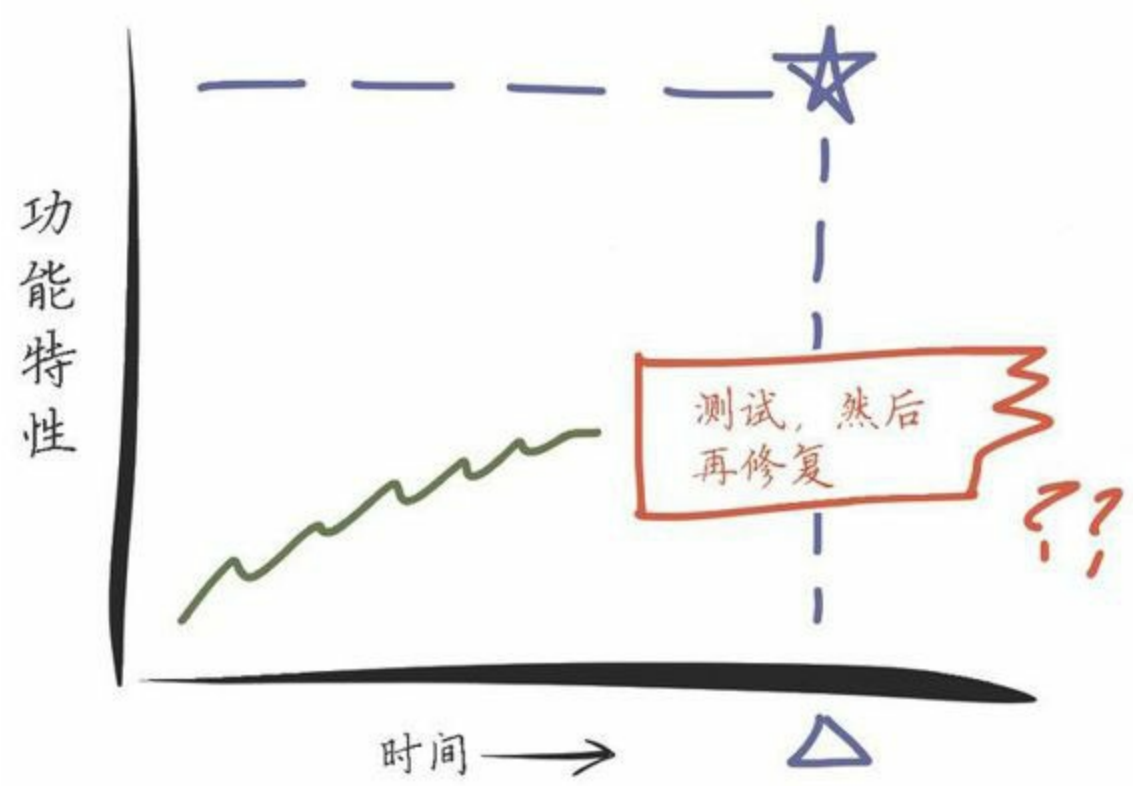
确定真实的进展。

根据功能特性构建产品这种方法使得每一次迭代都包含完整的开发流程：从需求到设计，然后编码，最后测试。这几个步骤总是会出现，因而也总是可见的。这种方法很安全、实用，而且富有效率。它将很适合你。

随着项目的进展，我们需要学习区分表面的进展与真实的进展。我们会逐渐理解，在特定的环境下什么样的“完成”才是真正的完成。

当看见真正的、可以运行的功能特性时，我们就可以对项目当前的状况有清楚且可靠的了解。反之，要是看不见能够运行的功能特性或者这些功能特性还没有完成，我们对项目的状况就会知之甚少。

当你的团队在开发功能特性一段时间之后告诉你说工作完成时，有哪些因素会使这些赞扬性的报告带来误导性的乐观效果？怎样才能够使这些报告更加准确无误？



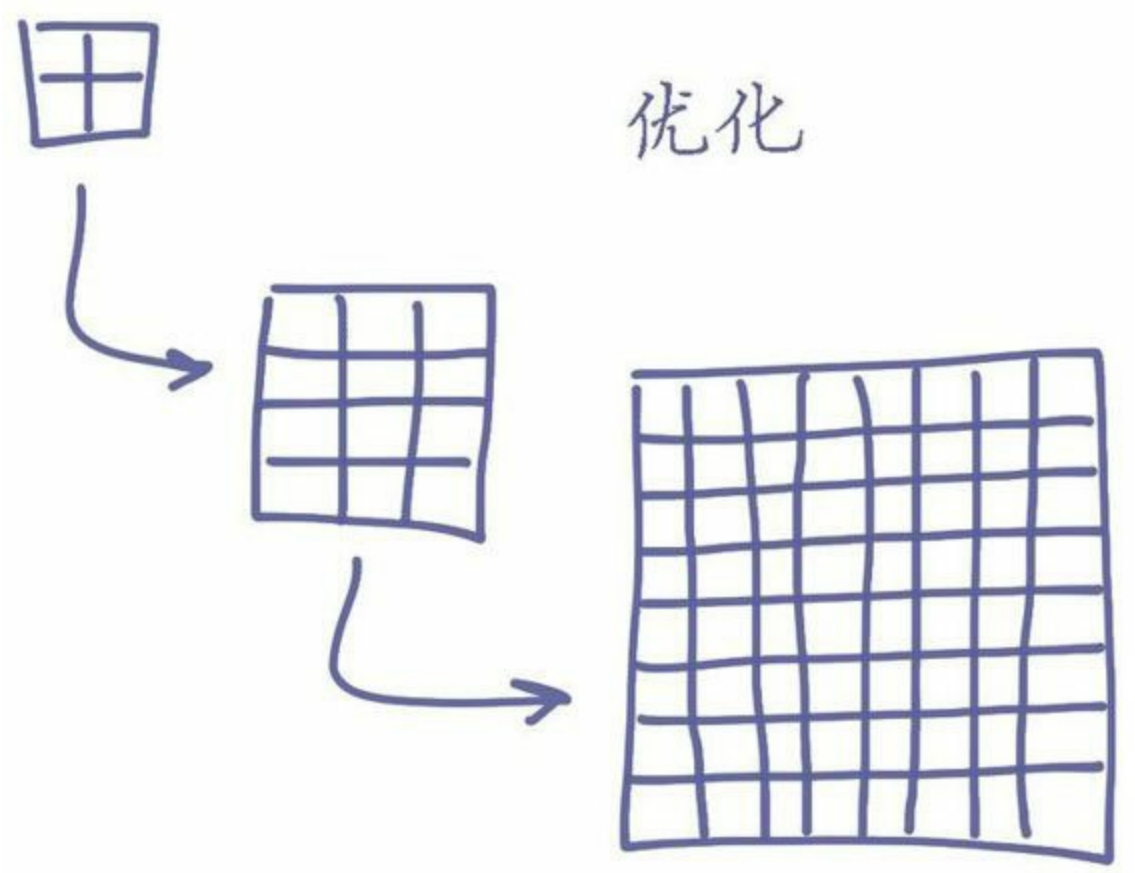
淘汰测试再修复式的收尾方式。

很多项目都会以测试修复、再测试再修复的方式收尾，而且这样的过程好像可以一直反复进行下去。这种收尾方式实在令人沮丧。即使采用了根据功能特性构建产品的方法，若功能特性并没有真正完成，这种情况依然可能会发生。

既然我们是在学习怎样在为期一到两周的迭代期内开发出真正完成了的、可交付的软件，就需要学习如何缩短或消除在很多项目收尾时都会出现的测试修复阶段。这个阶段持续的时间长，而且具有不确定性。

为了使根据功能特性构建产品的方法能够发挥作用，在为期两周的迭代结束时，我们开发出的软件必须是几乎没有缺陷的，而且必须总是这样。

为了确保没有缺陷，必须随时对一切进行检测。这并没有你想象的那么难，后面的章节将会详细讨论这一点。但现在，我们还是来看看本章的最后一点内容。



随着项目的进行，进一步扩展和优化设计。

随着我们完成的功能特性越来越多，只是没有缺陷是不够的。我们还需要对设计进行扩展。如果设计得太多，我们就没有足够的时间去完成那么多的功能特性，项目就会进展缓慢。然而，如果设计得太少，那么功能特性的实现就会比较难，进展同样会比较缓慢。而进展的快慢清晰可见。

通过观察开发速度的变化，也就是交付功能特性的速度的变化，我们就能学到什么程度的设计恰到好处。设计得太多，进展会缓慢；设计得太少，进展同样会缓慢。而我们的应对方法则是：调整，观察，再调整。

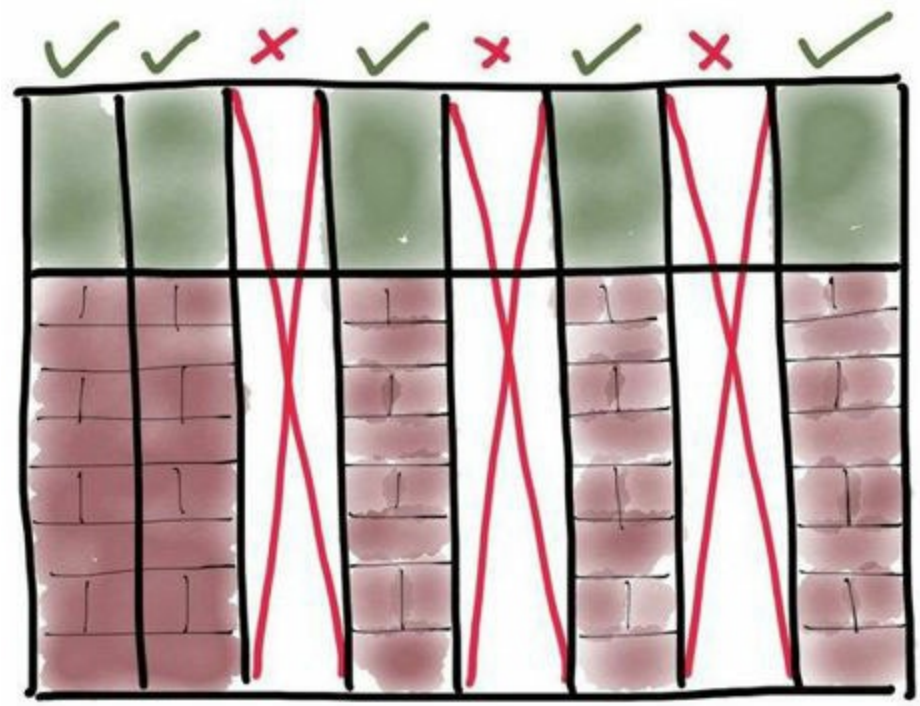
这是一项棘手的工作。很多人都见过设计太差导致很难取得进展的软件。好在保持软件的设计足够好的技术简单易学。不过，这些技术也很容易被遗忘，特别是在整个团队所承受的压力较大时。

那么，怎样才能够帮助开发人员保持设计整洁呢？他们是否掌握了必要的技能？应该避免哪些会影响到整洁设计的因素？

进阶阅读：

- 第18章　能力是提高速度的前提
- 第19章　重构

第7章 同时构建功能特性与基础



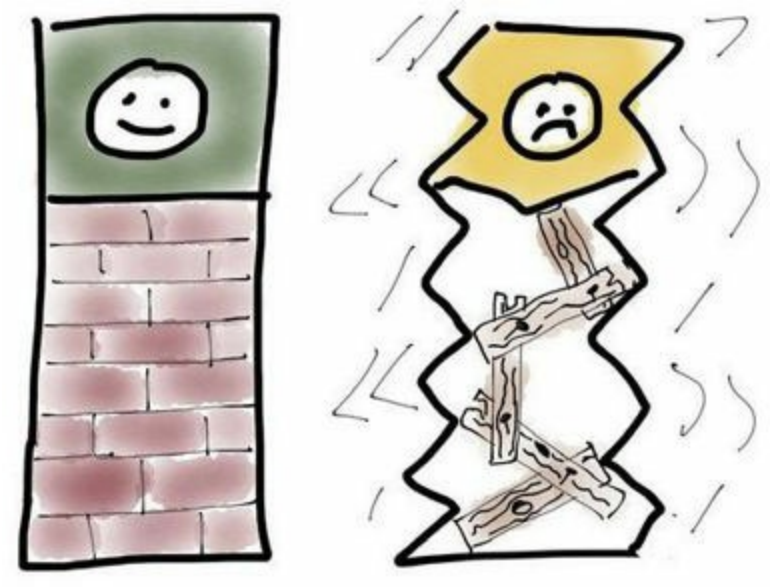
产品需要有坚实的基础

每一个产品都必须通过一组关键的功能特性来实现价值的交付。我们的目标是开发出这些功能特性，而将其余的去除。

无论是建造什么，都需要先打下坚实的基础，软件开发也不例外。在讨论基础时，我们会使用诸如“结构”“设计”或者“基础架构”这类的词语。

根据本书所阐述的指导原则，我们将采用增量的方式逐步构建产品的功能特性，优先构建最有价值的功能特性，而价值最低的则放到最后。为了确保各个功能特性都能够顺利地完成，从项目开始到结束，都必须保证系统的设计坚实可靠。

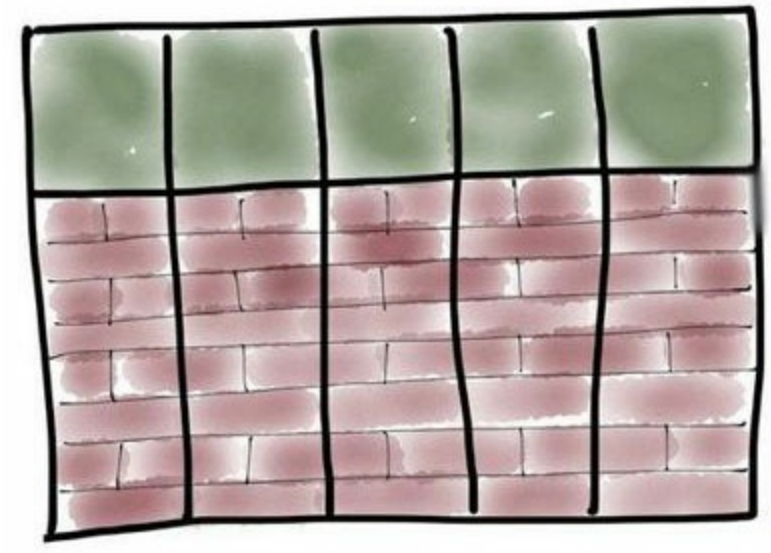
随着项目的进行，怎样才能在设计工作与功能特性开发之间保持最佳的平衡？



每个功能特性都需要有坚实的设计基础，或者说坚实的“基础架构”。

如果没有良好的设计基础，那么构建出来的产品肯定会到处都是缺陷，同时开发工作会变得很艰难。如此一来，进展缓慢自然不可避免，甚至连整个项目都很可能会失败。毫无疑问，功能特性的构建需要有基础的支撑，而坚实的基础需要有良好的设计。

没有基础，构建功能特性就无从谈起。那么，在必须拥有坚实基础的情况下，构建功能特性的最优方法是什么？

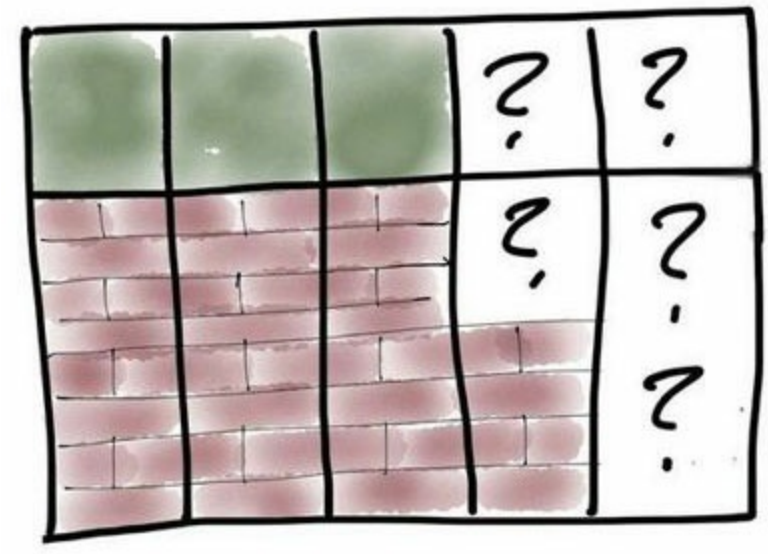


理想情况下，我们能够在项目截止时完整地交付所有功能特性。

不能忘记，在展望功能特性时，我们可是想要所有能够想到的功能特性。然而现实是，要拥有所有想要的功能特性几乎是不可能的。

即使在按优先级排序之后，需要实现的功能特性还是太多！为了能够在交付日期到来时提供最好的产品，必须完成的工作越少越好……而且必须要以最可靠的方式完成这些工作。

因此，我们必须要以具有下述特点的方法来同时构建功能特性和基础，即：安全要保证，速度也要快，同时还能够节省时间和成本。考虑以下两种方法：一是首先完整构建出项目的基础结构；二是在构建每一个完整的功能特性时也设计它的基础。这两种方法都有各自的缺点。



基础优先意味着能够进入市场的功能特性太少。

若优先构建基础，则在交付日期到来时，可以交付的功能特性几乎总是太少！

我们不会知道自己的开发速度能有多快，而且永远也不会知道。优先构建基础将导

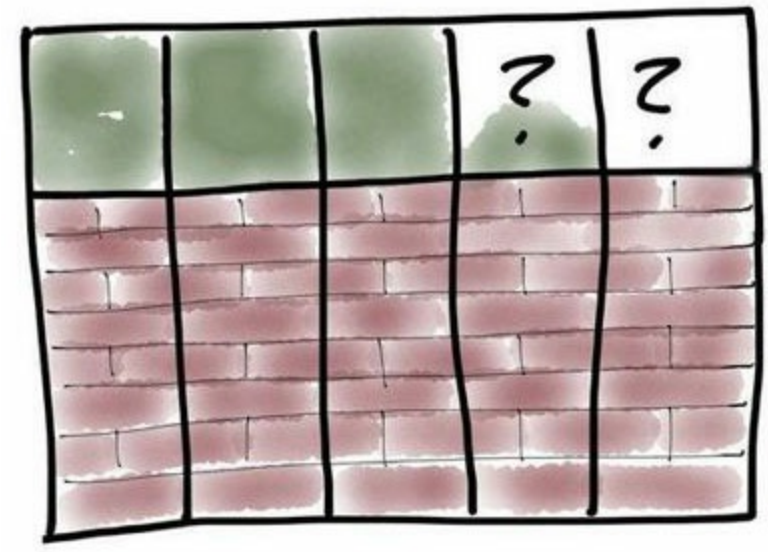
致我们构建太多的基础。因为构建基础消耗了太多时间，所以最终可交付的功能特性比我们本来可以交付的要少。

每个功能特性，对于用户来说都意味着价值，对于我们自己来说则意味着收入或者其他收益。只要有可能，功能特性当然是越多越好。

即使我们有足够的时间先去完成所有的基础工作，再在这些基础之上构建功能特性，我们依然只能在看见功能特性可以实际运行时才能够对项目进行指导。也就是说，优先构建基础妨碍了我们管理项目的能力。

我们并不敢采用优先构建基础的方法，因为它会在延缓项目进度的同时减少产品的价值。

那么，在构建功能特性的同时也设计其基础的方法又怎么样呢？



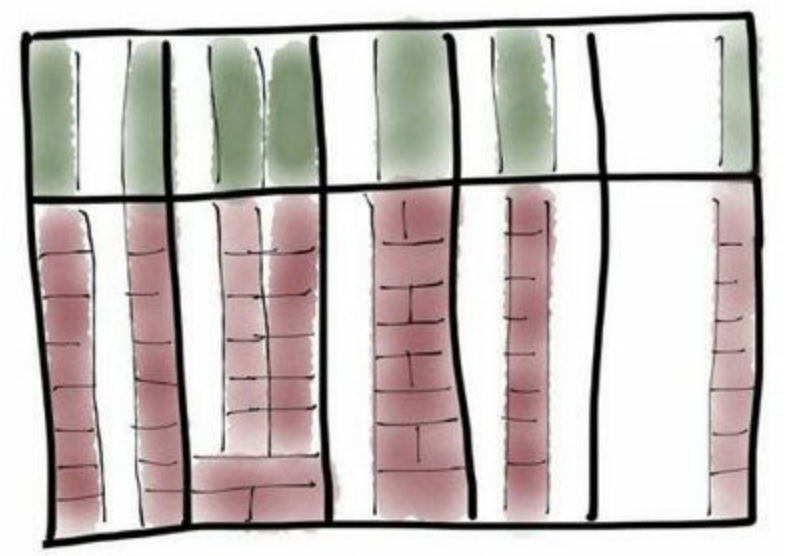
每次构建完整的功能特性（包含基础）同样意味着能够进入市场的功能特性太少。

如果逐一构建完整的功能特性，当交付日期到来时，产品很有可能会缺少关键的功能特性。

当展望某个主要的功能特性时，我们想尽可能完美地实现它，就像我们展望产品时想让它拥有所有可能的功能特性一样。

正如构建所有可能的功能特性是错误的，一开始就将每个功能特性实现到极致同样是错误的。用户是否对产品有购买意愿，取决于它的功能特性组合是否完善到足以吸引和满足他们。

既然不能优先构建基础，也不能优先完成所有的功能特性，那么还有什么其他的方法？难道说我们注定会失败？



首先构建简单而实用的版本。

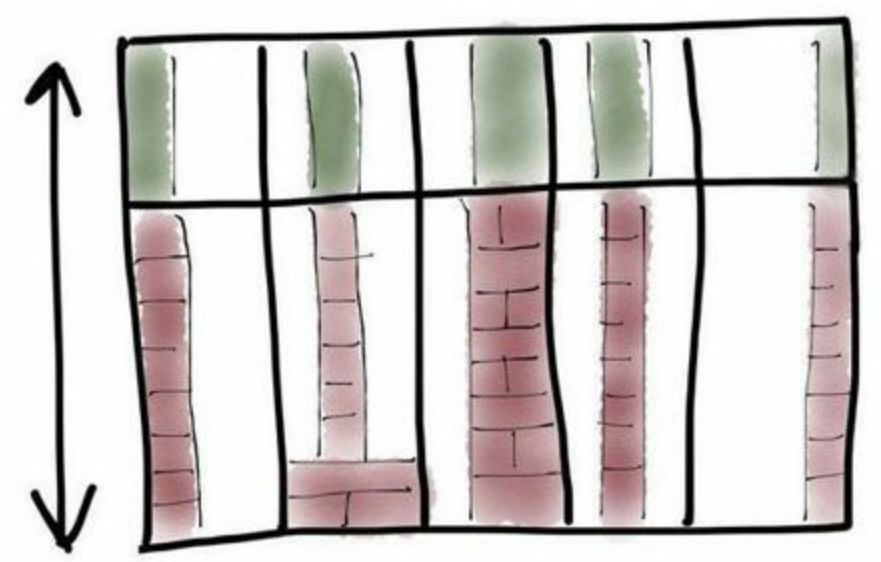
与前两种方法相比，首先为每个功能特性构建简单而实用的版本要安全得多。

为了能在有限的时间里构建出最好的产品，我们需要实现所有对于用户来说重要的功能特性，而不仅仅是其中的一部分。在时间允许的情况下，要最大程度地实现每个功能特性。这需要我们自己判断用户需要什么以及我们还剩多少时间。

如果对于每个必需的功能特性，我们都能构建出小的版本，同时还为它们打下足够

坚实的基础，那么我们就能够做到最好。

这样一来，我们就能以最快的速度构建出最小可行产品（minimum viable product）。

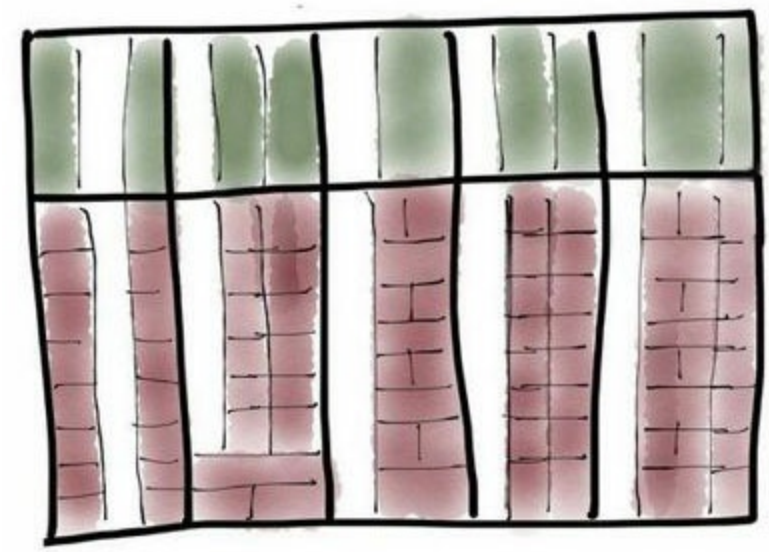


在多次迭代中逐渐完善每个功能特性。

首先判断什么是最重要的，然后逐渐完善每个功能特性，并随着项目的进行，构建刚好够用的基础。

与其碰碰运气，看看在交付日期到来时能够完成多少基础或功能特性，不如构建小的功能特性版本。每个新的小版本都会使整个产品更加完善，这样我们在每一个时间点都会拥有当时所能拥有的最好的产品。

在每一次迭代中，我们都可以重复这一过程，而且随着项目的进行，可以随时调整每个功能特性的优先级，直到时间和成本告诉我们是时候停止了，或者该去关注新的产品了，正如第2章所述。



为在交付日期到来时获得尽可能好的结果而努力。

由于我们在每个时间点都拥有当时所能拥有的最好的产品，因此当我们决定交付时，所拿出的是交付时最好的产品。实际上，由于我们一直都处于可以交付的状态，因此，只要有提前交付的理由（这种情况很常见），就能够做到这一点。

这的确需要能力！

如何选择功能特性的最佳组合，需要产品推动人有很强的能力。不过，这一能力是可以培养的，而培养的最佳方法就是反复选择下一迭代要开发的功能特性。勤勉的产品推动人将会看到他的产品愿景变成现实，同时也会在反复的选择实践中学着去做最佳决定。

同样，开发人员也需要有相应的能力。开发人员经常被要求预先设计系统。正如我们所见，这并不是理想的方法，因为我们从来不会预先知道某个系统最终会是什么样子。当团队的产品推动人仍在探索他需要什么、能够获得什么以及怎样做出最好的系统时，情况尤其如此。

那么，为了能够以这种方式进行开发，整个团队需要具备哪些能力？业务人员又需

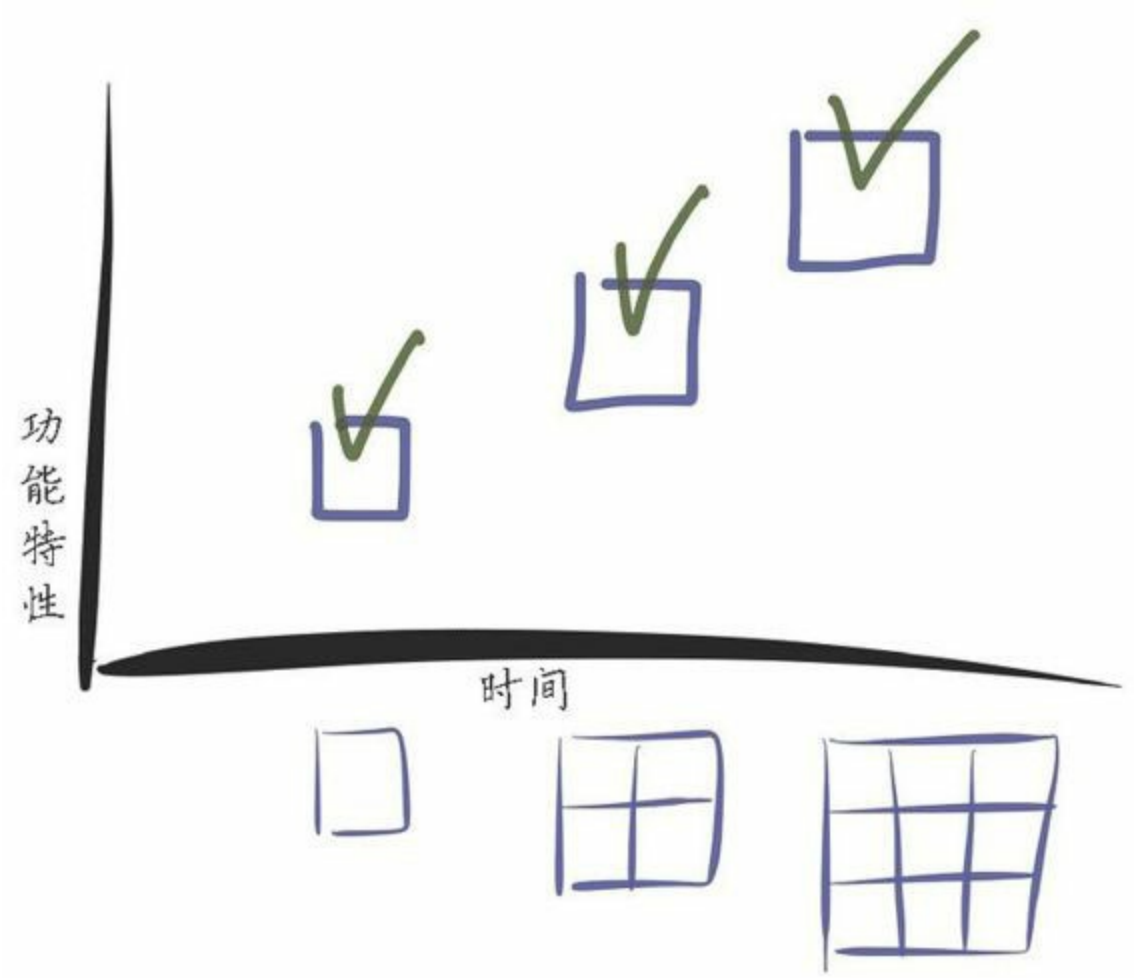
要提供哪些指导？如何帮助技术人员做好本职工作？

第8章 零缺陷与良好的设计

良好的过程控制可以有效地减少代码中的缺陷

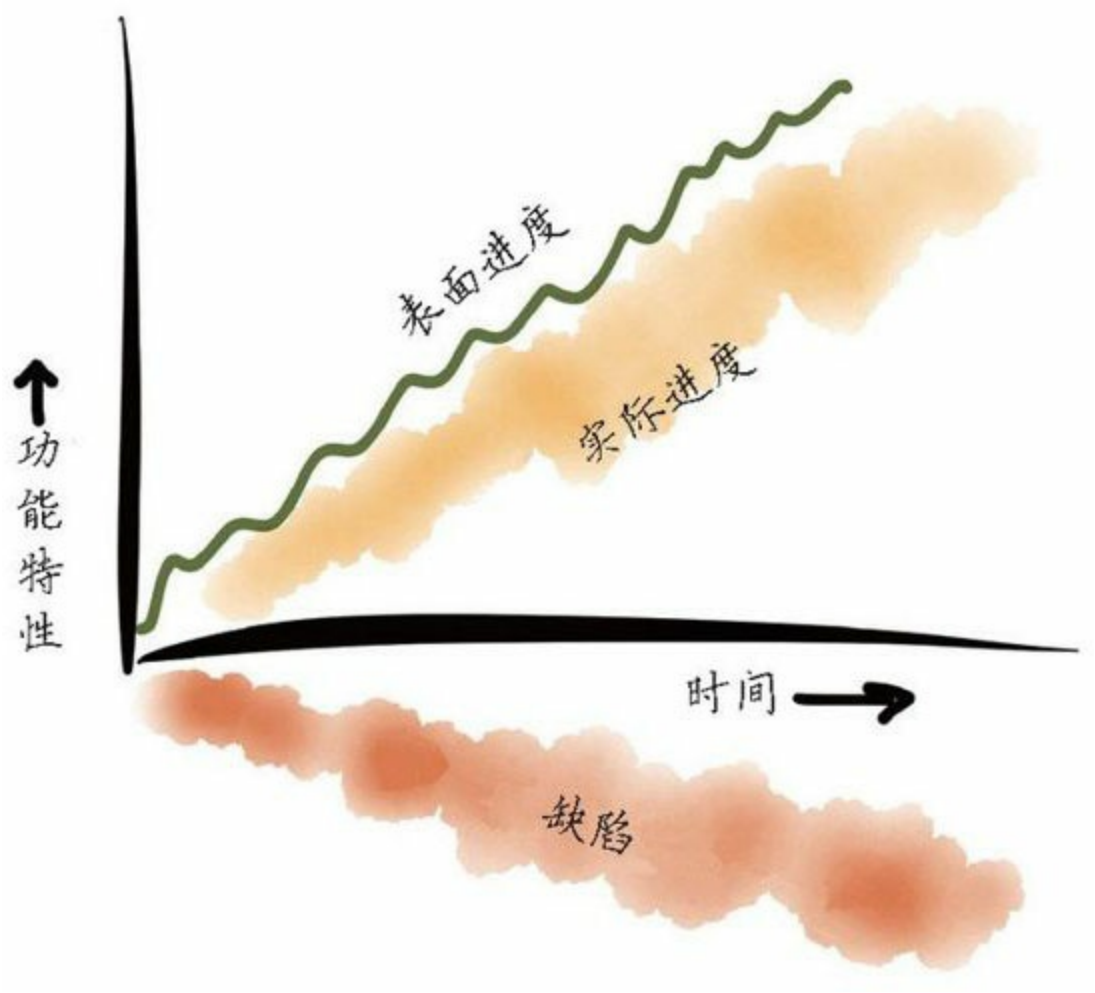
为了确保产品在任何时候都拥有良好的设计而且没有缺陷，我们需要有良好的技术实践方法。那么，什么是良好的技术实践方法呢？

让我们更详细地了解一下为了能够根据功能特性进行开发，开发人员都需要做哪些事情。不用担心，我们并不会深入到编程这个层面，而只是希望你意识到企业对开发人员的期望及需要提供的支持。



产品构建在不断发展和变化的设计基础之上，并由数量不断增长、能够正确运行的功能特性组成。

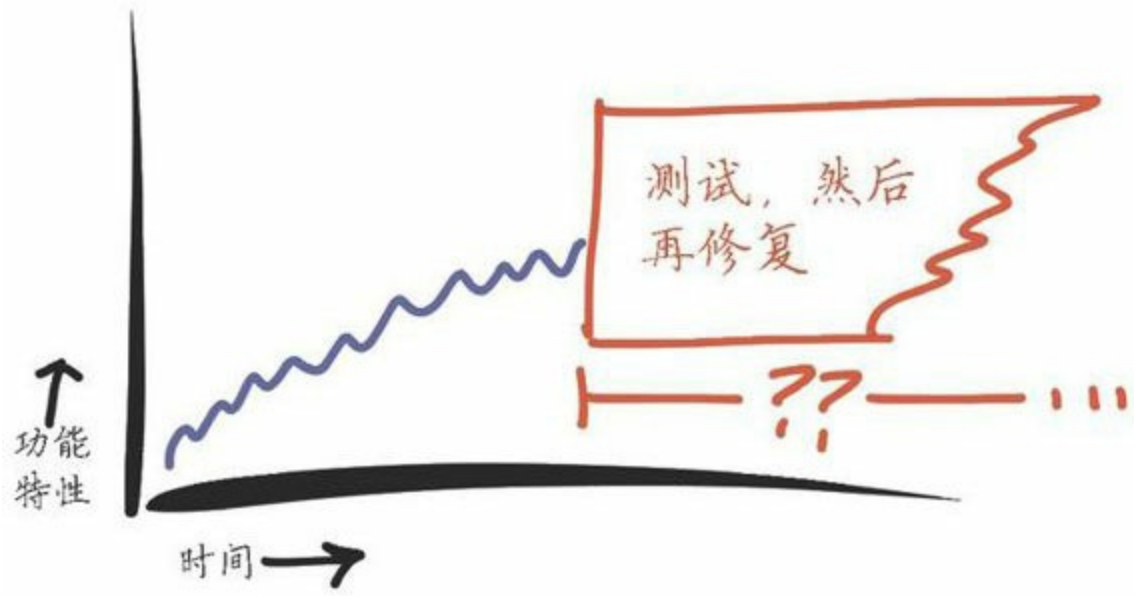
我们要求团队根据功能特性有条不紊地进行开发，保持一切都经过检验而且能够运行，并且随着项目的进行不断地完善其设计。这是引导项目走向成功的最佳机会，但其中仍然会有不少问题。



缺陷相当于拙劣的功能特性。它使项目进展变得不确定。只有消除缺陷，我们才清楚真正完成了哪些功能特性。

我们设法根据功能特性做计划和进行开发，同时对项目进行管理。这就意味着，当我们被告知某个功能特性完成时，它就是能够真正工作的。其中存在的任何缺陷都将使它变成拙劣的功能特性。一旦存在缺陷，项目进展就会变慢。如果还存在没有被发现的缺陷，那么情况会更糟糕：我们甚至都不知道实际情况会有多坏。

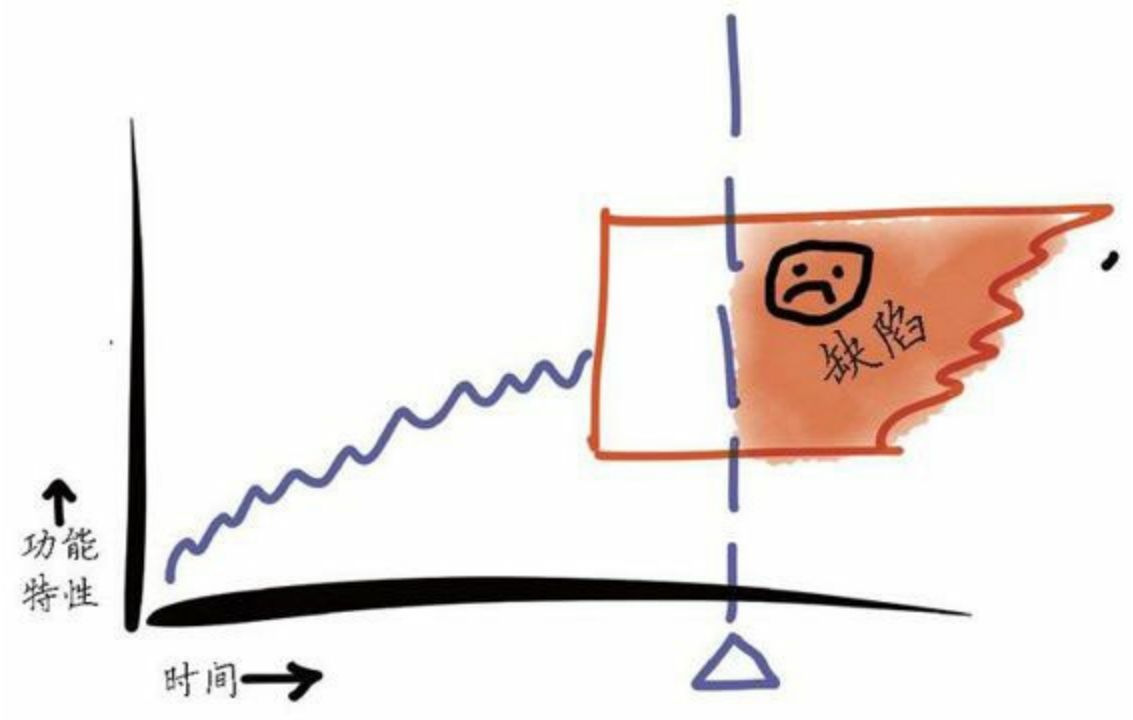
在一个充满缺陷的世界里，我们不可能高效工作。



修复缺陷会带来不确定的时间延迟。随时发现缺陷随时修复，这样才能清楚知道完成了哪些功能特性。

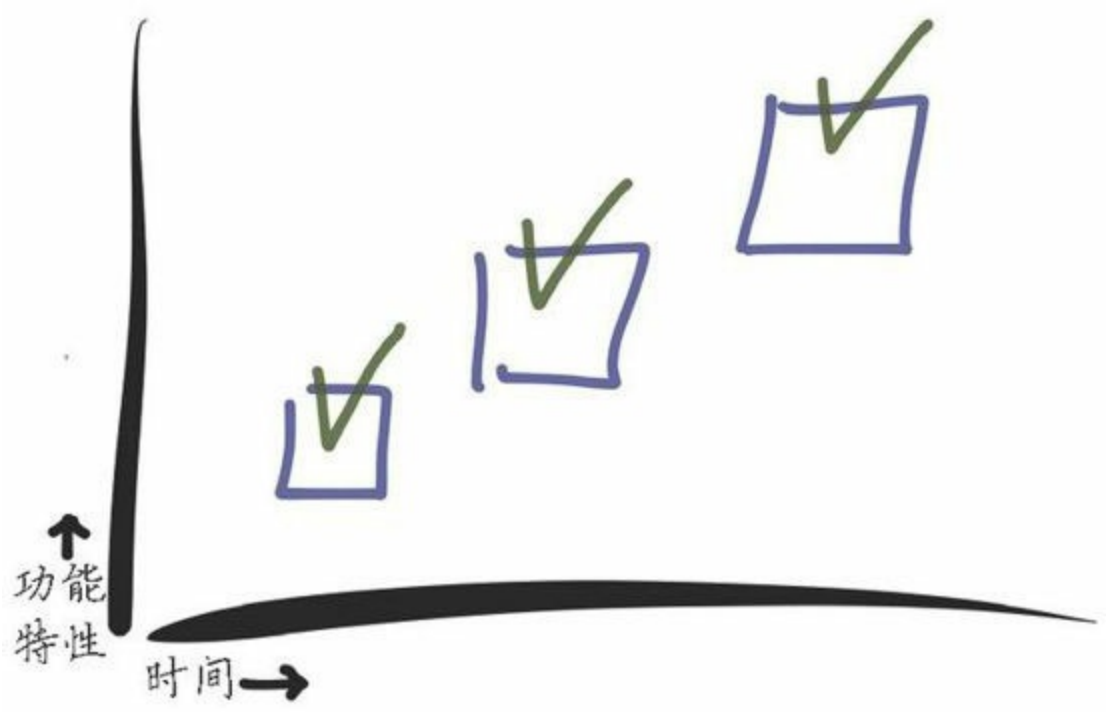
正如咨询师汤姆·狄马克（Tom DeMarco）所说，没有人会在看见餐厅地板上有一只蟑螂时说：“瞧，有一只蟑螂。”对于缺陷而言，情况也一样。如果出现了一个缺陷，那就意味着可能会有很多缺陷。

找出缺陷将花费很多时间，修复这些缺陷同样也要花费时间。如果将这些工作留到项目快结束时处理，那么开发新功能特性的计划就会被打断，同时我们还必须决定暂不修复哪些缺陷。



如果不能及时知道已完成哪些功能特性以及完成的质量怎么样，那么就只能延迟交付，而且交付的是明显有缺陷的产品。这是很糟糕的商业行为。

“进行缺陷类选如何？”用不着，谢谢。我们必须尽最大的努力避免缺陷，因为它会在项目的收尾阶段增加不确定的修复时间。延迟交付产品，而且交付的还是有缺陷的产品，这会让我们看起来很蠢。我们最好还是别这样。

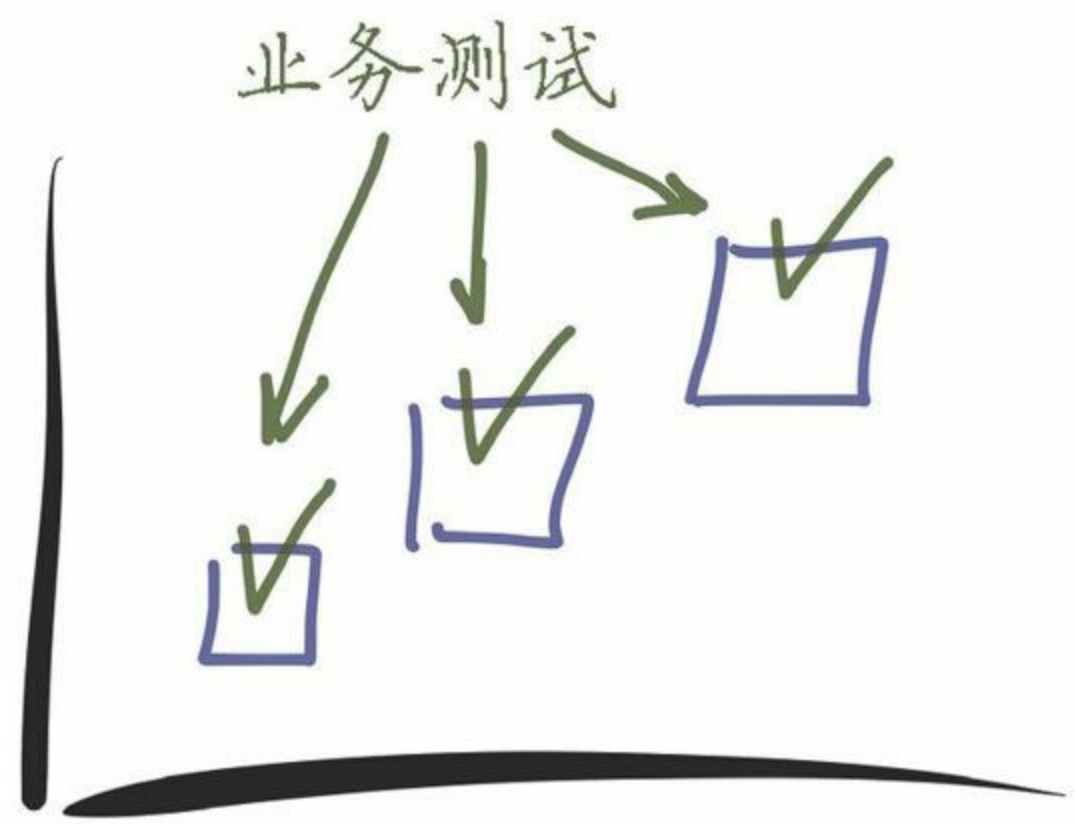


因为功能特性是逐渐增加和完善的，同时其设计也是逐渐改进的，所以出现错误在所难免。我们需要进行持续的、全面的测试。

实际上，并没有什么别的好办法。在每一次迭代结束时，我们都必须尽可能地使软件没有缺陷。要做到这一点，唯一的方法就是测试。

由于系统的功能特性在不断增加，因此我们需要越来越多、越来越细致地测试。不仅新的功能特性需要测试，旧的功能特性同样也需要测试，以确保它们没有因为新功能特性的开发而受到破坏。

我们需要进行两个层面的测试：“业务”测试和“开发人员”测试。

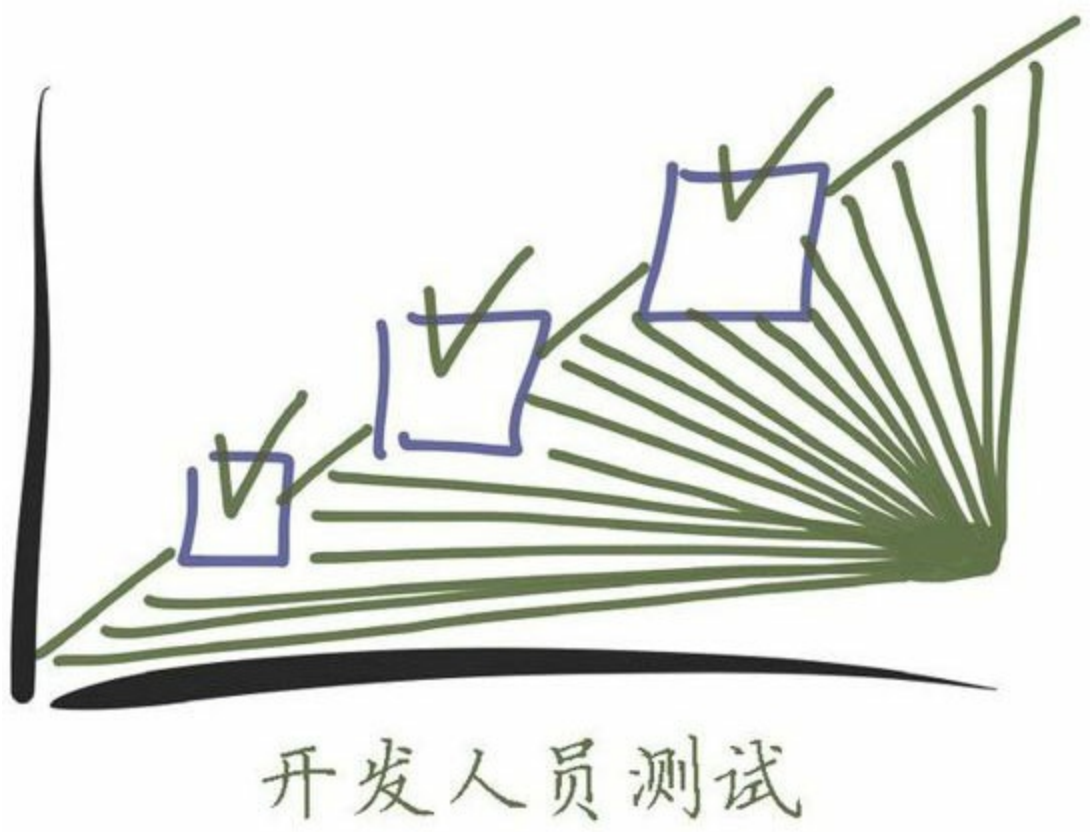


在每一次迭代结束时，都需要进行业务层面的测试来验证是否得到了想要的软件。

产品每两周就会有新的功能特性加入。我们必须知道这些新的功能特性是否有效，同时也需要确保旧的功能特性没有受到破坏。为了实现这一目标，必须进行业务层面的测试，并且尽最大努力验证每个功能特性的各个方面。

如果不进行验证，就不知道功能特性是否有效。因此，必须进行各方面的验证。每增加一个新的功能特性，测试的任务量也会随之增加。我们必须要跟上测试任务增加的节奏。

目前已知的最好方法，就是表述功能特性必须通过哪些测试，并且使这些测试工作自动化，以确保该功能特性不仅现在可用，而且以后一直可用。这就是人们常说的验收测试驱动开发（acceptance test-driven development，ATDD）。

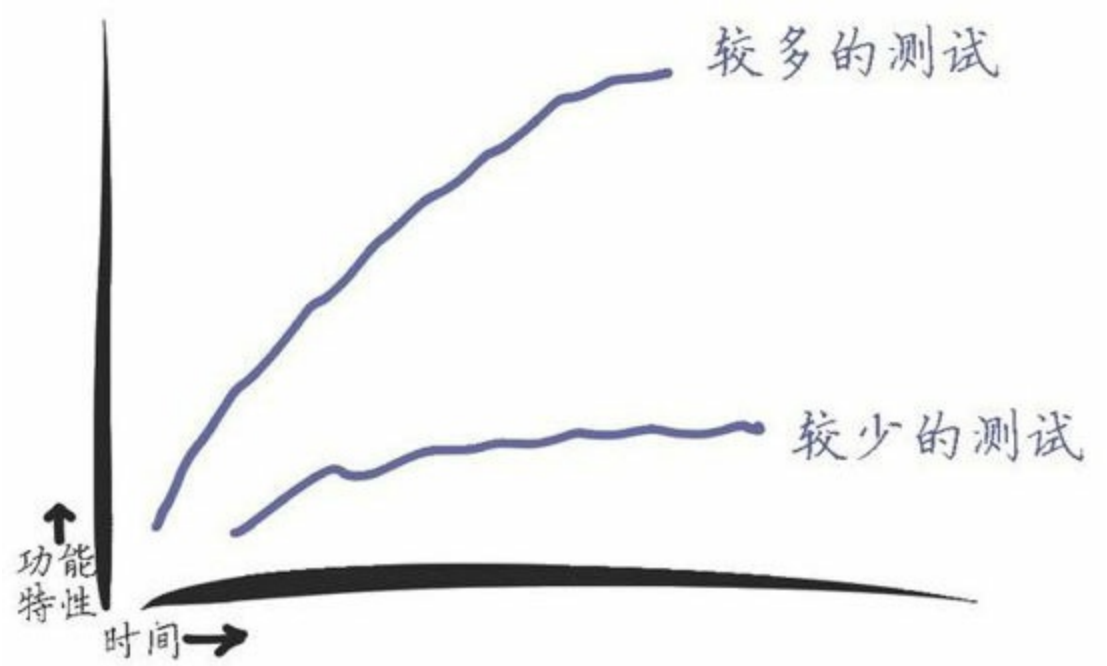


开发人员每天都会改动代码。为了保证他们不浪费时间，开发人员测试需要以更快的频率进行。

一个功能特性需要用几百行甚至几千行代码实现，其中任何一行代码都可能出错。果真如此的话，该功能特性肯定会在某种情况下出错。

我们可以依赖业务层面的测试发现这些问题，不过那通常需要花费很长时间，而且即使发现了某个问题，通常也不会指出具体的编程错误。因此，开发人员需要构建综合的自动化测试网络，以确保尽早发现和解决问题。

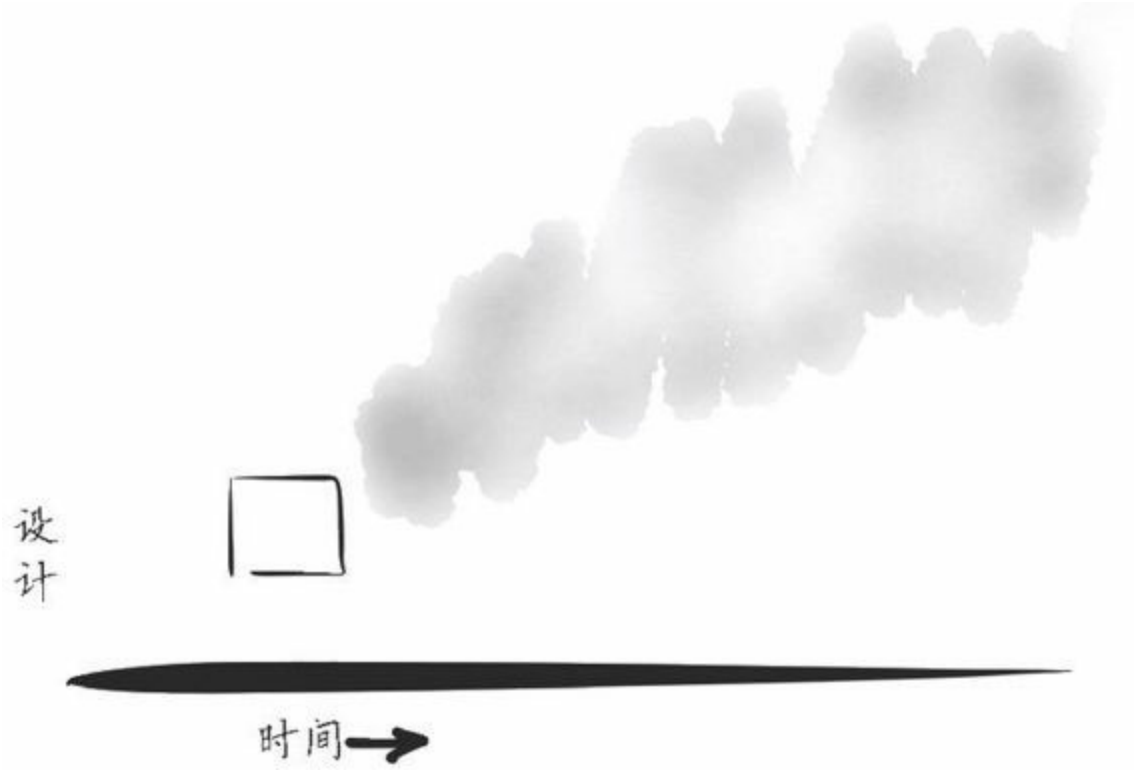
要实现这一目的，已知的最好方法就是先写测试代码，然后再运行这些代码。这通常被称为测试驱动开发（test-driven development，TDD）。它最大的优点在于支持对设计的改进，这一点留待稍后讨论。



奇怪的是，测试不但没有减慢开发速度，反而使其变得更快！这是因为测试可以使我们犯更少的错误，同时使错误更快地被发现。

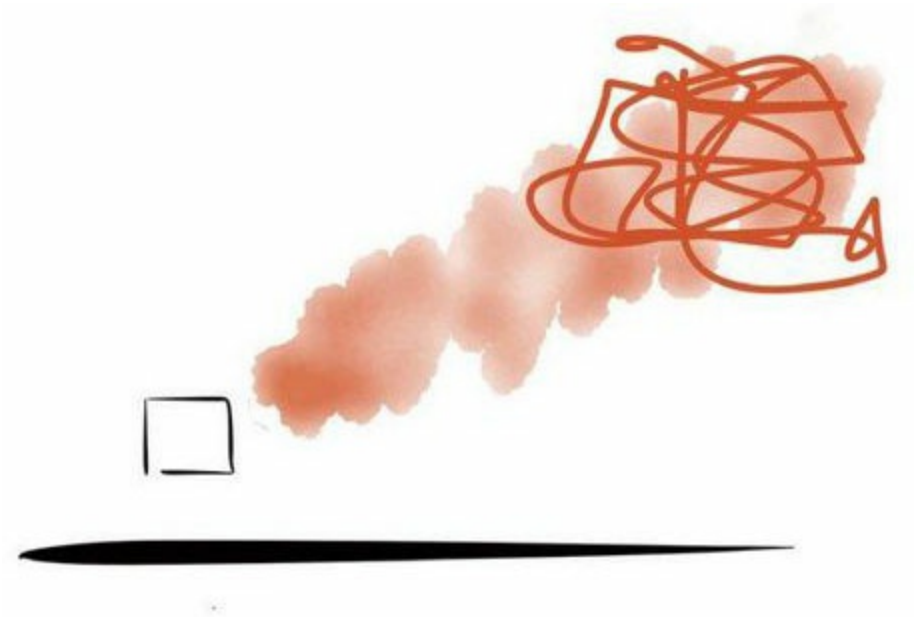
与从一开始就避免问题产生相比，找出并解决问题所需的时间会更长。当进行业务层面的测试时，如果功能特性没有按照需求通过测试，那么就不会被发布；当进行开发人员层面的测试时，测试用例是在编写代码之前或者是与代码一起完成的，如果测试不通过，那么我们同样也不会发布软件的任何部分。

作为开发工作的一部分，测试可以阻止缺陷在程序中出现。这比直到我们认为编码已经完成时才进行测试要快得多。



最开始时，软件只有一些比较小的功能特性，此时的设计可以比较简单。

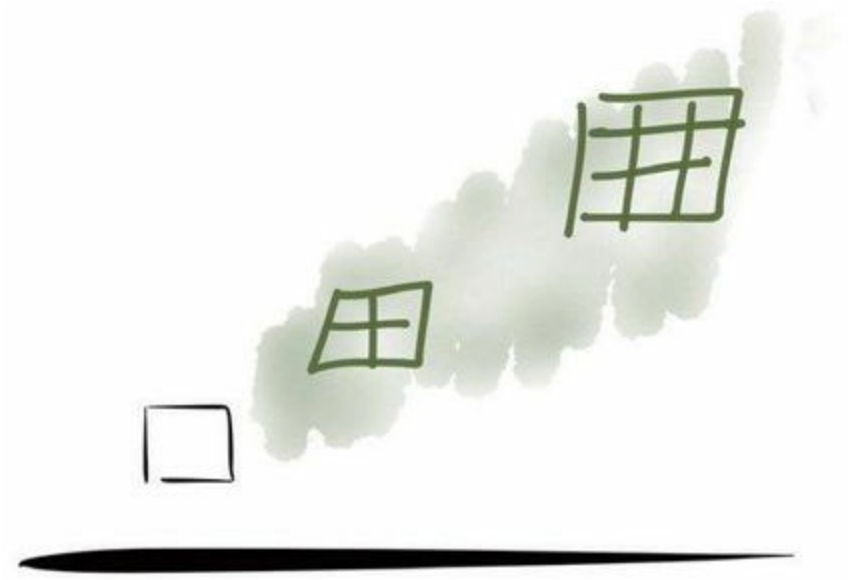
记住，我们进行的是增量式开发，每两周就会发布一些真正可用的功能特性。因此，从一开始我们就需要有良好的设计，只不过此时比较小的设计就足够了。



设计是很容易退化的。

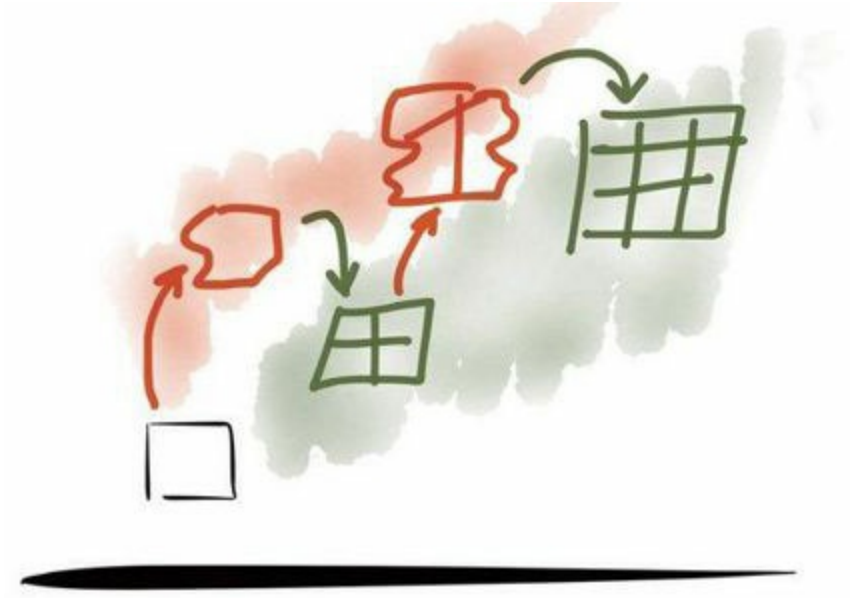
当我们向软件中增加功能特性时，即使软件拥有很好的设计，情况也会变得糟糕起来。我们填填这边的窟窿，又补补那边的漏洞。这样经过一段时间以后，再好的设计也会变差。虽然可以避免这样的情况发生，但这需要我们拥有娴熟的技术，并且能够随着项目的进展随时对设计进行调整和优化。

拙劣的设计会减慢开发速度。为了维持项目的活力，能力和细心的态度都是不可或缺的。



随着系统的功能特性越来越多，设计也需要进行相应的扩展。

任何时候，我们都需要有良好的设计。拙劣的设计之所以会减慢开发速度，是因为要想改变它很难。随着项目的进展和功能特性的增加，我们必须对原有设计进行相应的扩展，以便支持新的功能特性。总之，任何时候都需要有高质量的设计。



在构建功能特性的每个阶段，为了保证设计对功能特性的支持，项目团队必须对设计进行足够的改进。

每处改动都可能会破坏当前的设计。为了保证设计一直处于良好状态，我们需要随着项目的进展不时地改进它。改进措施可能是为新的功能特性预留足够的空间，也可能是先添加功能特性，然后再重新处理设计，以保持良好状态。这两种方法都可以。放任设计慢慢地退化则是不可取的。

在改变设计的同时保持其良好状态，这通常被称为重构（refactoring）。在进行增量式软件开发时，重构是一项必备技能。关于重构，我们还会在其他章节进行进一步的讨论。而现在，让我们来重点关注其优点。

若不能保持设计处于良好状态，轻则影响项目的进度，重则导致项目失败！

如果放任设计退化，项目的进度肯定会变慢，增加功能特性的成本也会大大超过设计处于良好状态时的成本。除非一直使设计处于良好状态，否则开发速度就会越来越慢。

测试与重构结合在一起，使得增量式开发成为可能。软件开发工作的本质要求我们进行测试和重构。到目前为止还没有发现更好的方法。

业务人员需要确认软件可用，验收测试驱动开发就是一种随时了解哪些功能特性可用的好方法。开发人员需要精确地知道问题出在何处，测试驱动开发则是达成这一目标的主要工具。强大的自动化测试套件让我们可以随时知道哪些功能特性可用，而两个层面的测试，即业务测试和开发人员测试，则能让这项工作完成得更好。

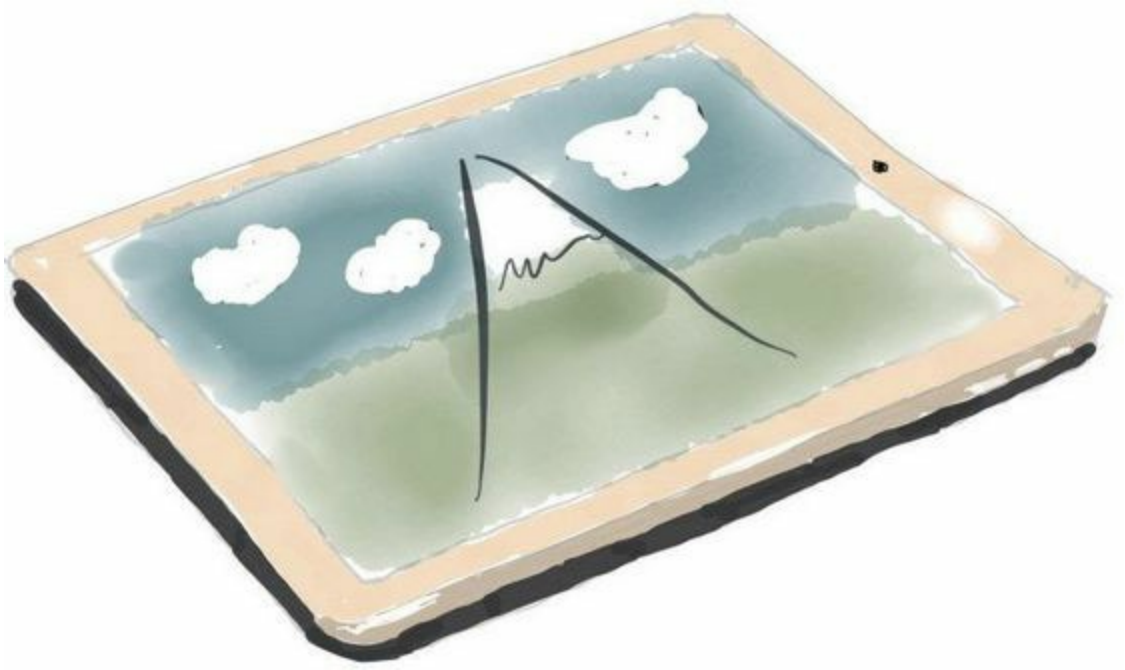
项目若要进展，就必须持续改进其设计，而利用重构对设计进行改进和优化则是实

现这一目标的方法。然而，即使是最好的重构也或多或少会产生一些错误。业务层面和技术层面的测试则使我们有足够的信心，来进行所需的设计改进与优化工作。

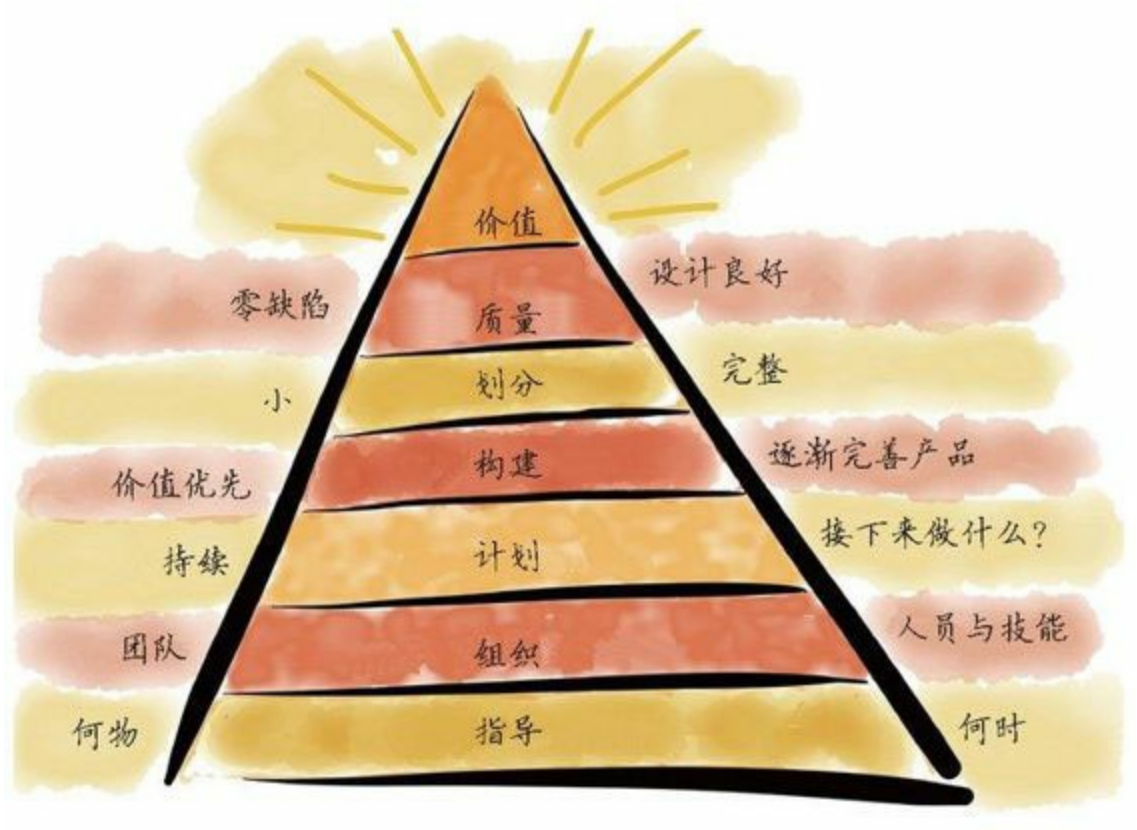
这就是目前我们所知道的能够保证产品质量良好、流程进展顺利、进度可以预见的最佳开发方法。测试与重构是这种方法最重要的两种工具。请别忘记使用它们。

进阶阅读：

- 第14章　组建强大的团队
- 第17章　监督员工更加努力地工作
- 第18章　能力是提高速度的前提
- 第19章　重构



第9章 价值的完整循环



让我来解释一下。算了，内容太多，我还是来总结一下吧。

（1）我们最终想要的是价值，提供价值的则是功能特性。功能特性发布得越早，我们就能越早提供价值。

（2）基于价值的管理比基于时间或工件等不提供价值的事物更胜一筹。

（3）根据功能特性做计划很简单，只有在必要时才进行估算。根据以往完成的工作量来安排下一阶段的工作，效果会更好。

（4）采用逐渐增加功能特性的增量式开发方法，要求我们每隔几周就能够开发出小而完整的产品。所开发的产品必须总是能够正常运行，而且其设计也是良好的。

（5）开发工作必须要交付真正可用的功能特性。产品必须经过严格的测试，业务人员和开发人员都需要参与其中。同时，产品还必须拥有良好的设计，而且开发人员需要保持产品的设计一直处于良好状态。

以上就是价值生产的全部过程，可以说很简单。为了能够获得最大的价值，需要上至公司高层，下至每个项目的管理人员和开发人员为之付出努力。让我们行动起来，共同打造有价值的软件吧！

进阶阅读：

- 第13章　事情并非那么简单

第二部分 说明与论述

正如第一部分所述，软件开发的基本流程很简单。把工作做好虽然简单，却并不容易，其中的细节更是无穷无尽。本部分将更深入地研究其中的一些细节。你将会读到对某些话题的简短说明与论述。我对这些话题饶有兴趣，希望你也如此！

第10章 价值是什么

有读者问道：“罗恩，你在第2章里说，价值就是那些我们想要的东西。难道你不觉得这样的表述过于简单了吗？”

我回答道：“是的，本书对所有内容的表述都很简单。” 然而，如果想让我多说几句以便提供更多线索的话，不妨读读第11章。

在《禅与摩托车维修艺术》一书中，作者罗伯特·波西格塑造的主人公斐德洛一直在探索什么是良质（quality）。斐德洛认为，“良质，就是那些你喜欢的东西”。

在本书中，我提出了一个类似的观点。

正如在其他许多领域中一样，我们在敏捷软件开发中也讨论“价值”这一概念。我们会基于价值决定要做什么，以及不做什么。我们会先做价值高的事情，而将价值低的事情放到后面去做。那么，这里的价值指的是什么呢？

简单来说，价值就是“我们想要的东西”。

对于这样的表述，你可能会认为它有点“禅”的意味，或者认为它根本就没有什么意义。接下来，我们将探讨价值一词代表什么意思，以及它怎样伴随着我们。我的目的是帮助你明白，价值事实上就是我们想要的、看重的东西。

敏捷方法要求我们基于价值决定做事情的先后顺序。有时候我们可能会说到商业价值或者客户价值，好像这些限定词可以让我们表述得更清楚。从某种程度上来说，这些词的确能够起到一些作用，因为它们会使我们从“在商业上有用”或者“对客户有用”等角度思考我们看重的东西。但是这些远不是我们所能想到的唯一的价值种类。让我们再多看一些例子。

我们可能会为产品选择一个战略方向。我们认为需要了解用户都喜欢什么，因此构建了几个原型并向潜在用户展示。

这里，我们看重的是信息。

我们的产品可能旨在挽救人们的生命，比如说它可以帮助我们将疫苗快速地运往世界各地。因此，我们决定根据每个功能特性所能够挽救的生命的数量，来确定接下来要开发哪些功能特性。

这里，我们看重的是人的生命。

我们的公司可能出现了资金周转困难。因此，我们决定去找一些风险投资，并快速开发出样品，以向潜在投资者展示。

这里，我们看重的是资金，是公司能够生存，是如果能够继续经营下去所拥有的帮助客户的能力。

我们的产品运行速度比较慢，顾客由于这个原因而选择使用其他类似产品。因此，我们决定推迟增加新的功能特性以提高软件的运行速度。

这里，我们看重的是产品的运行速度。

我们的进度可能很慢，完成全部功能特性需要花费的时间太长。所以，我们决定推迟增加新的功能特性以精简软件，这样一来，开发速度就能够更快。

这里，我们看重的是进度。

我们的产品可能会展示一些很可爱的猫的照片。这样做的目的是使人们露出笑脸，为他们的生活增添点滴幸福。

这里，我们看重的是人们的幸福。

我们看重快乐，看重创造力，看重合作。我们也看重金钱，看重收益，看重持续工作的能力。我们还看重与我们关心的人在一起，做我们喜欢做的事。我们看重人的生命，甚至还看重猫的生命。

有价值的事物太多太多，以上只是一些例子而已。产品负责人、管理人员以及所有决定下一步应该做什么的决策人员，要深入了解我们所看重的事物，然后决定开发的先后顺序，最终使我们在构建产品时所付出的时间、成本和精力获得最大程度的

回报。

如果事情能够更简单，譬如说“价值就是未来90天的收益”或者“价值就是销售副总裁所想要的东西”，那么事情就好办多了。当然，对于一些人、一些公司来说，这样的价值定义有时的确是适用的。但是，对所有人都适用的价值定义并不存在，而且在我看来，这样的定义也不可能对每个人都十分有效。不论是公司还是个人，要想生存下去，就必须深入了解价值，然后从可能去做的所有事情中选择那些最重要的去做。

选择价值，就是选择那些对我们重要的东西。

所谓价值，就是那些我们想要的东西。

第11章 如何衡量价值

第一部分从一开始就提出了“价值就是那些我们想要的东西”这一观点，并建议采用每几周就产生实用价值的方法进行软件开发。同时，要用查看实际软件的方法来了解软件开发的真实进展情况。关键在于，你需要重点关注价值，而不是成本，而且需要知道能够提供价值的只有真正可以运行的、我们了解其功能特性的软件。

可是罗恩，听你的意思，好像价值完全是主观的。难道我们不应该根据真实可靠的数据信息来进行决策吗？那么，你都有哪些反对衡量它的理由呢？

你说得对，我的确反对用数值方法衡量价值，即使是估算成本我也不赞成。下面，我来解释一下原因。

首先，我们并非真的知道数据。

对于几乎任何相关的产品，我们都不知道数据。我们既不知道有多少用户会使用功能特性，也不知道产品能够挽救多少人的生命，同样不知道人们会给我们最新的想法打五星还是三星。我们还不知道在打完分后，他们是否会购买产品。

其次，大的差别很重要，小的差别则并不重要。

当我们审视面前所有的功能特性选项时，会发现其中有一些功能特性非常重要，而另外一些则相当乏味无趣。这两者之间的区别很重要：我们需要区分出哪些是特别重要的，哪些是乏味无趣的。

最后，不同类型的价值不具有可比性。

产品推动人会关注很多不同类型的价值，而且随着时间的推移，某种价值可能变得

越来越重要或者越来越不重要。有时，我们需要知道用户的看法：他们会觉得某个想法有用吗？有时，我们需要了解与开发相关的信息：实现某个想法是需要花费一天、一周还是十年？还有些时候，我们更看重的是能否取悦用户或者潜在用户：如果能够快速地交付某个功能特性，那么他们可以马上付款。那么，到底该怎样做呢？

但是罗恩，“价值”到底是什么呢？应该如何衡量价值呢？我们怎样才能够确定得到了价值呢？

不必恐慌！其实，对于价值是什么，你已经很清楚了。不妨挑两件你可能要做的事情，问一问自己接下来会去做哪一件。你选择做的就是当前更有价值的事情。你几乎总会知道要去做哪件事情。若不知道，就再问一问自己是否这两件事情都不值得

去做。通常，你会发现它们并不值得去做。

确定做哪件事情之后，再来想想你为什么会这样选择。将你的想法记下来，然后继续追问为什么。通过这样做，你就能认识到你关注的是哪些维度的价值。然后咨询项目干系人，问问他们的想法及其理由。这样就能了解什么以及为什么重要。

我们总是很难抗拒用数值表示价值的做法，如果你的确有一套这样的表示方法，不妨试一试。挑两件事情，计算一下两者的价值所对应的数值，看看所得的结果是否与你的想法一致。若一致，就继续使用这一方法；若不一致，那就很有趣了！继续完善你的方法，直到两者一致。如果总是不能一致的话，我建议你干脆放弃这一方法。

寻找数值来表示价值可能会使我们滑入深渊。如果公司开发产品的目的是赚钱，那么就可以用产品的销售收入来衡量它的价值。但是实际上，并没有什么好方法来判断这一数值是否是不错的衡量指标，因为它将销售问题、产品问题，当然还有顾客的问题都混在了一起。

更糟糕的是，大多数与金钱有关的衡量指标在时间上都落后，因为等我们得到有关的信息时，为时已晚。同时，也没有真正好的办法来判断收入数据是好是坏。因此，金钱是槽糕的指标：不仅慢，而且还无法判断它是好是坏。

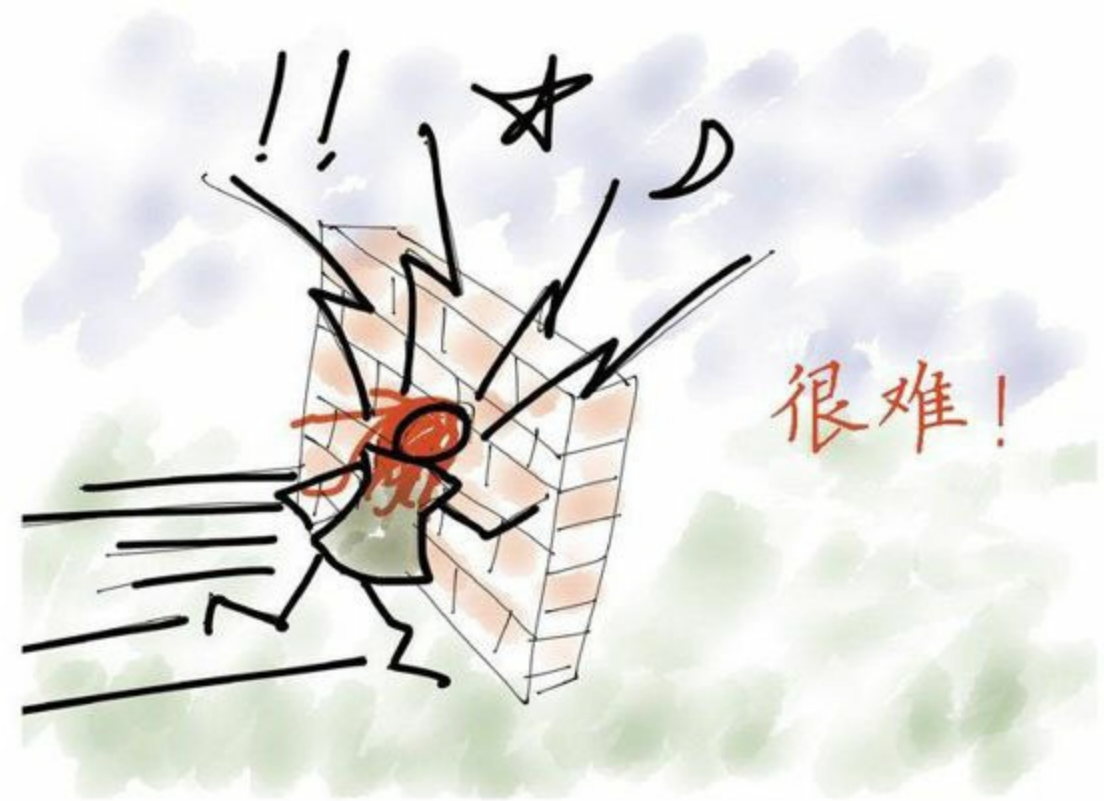
我很希望能够有一个简单的答案，像功能点的数量，或者用户点击次数等。如果你发现这些数值有用，不妨继续使用。不过，需要明白的是，所有这些衡量指标的真正作用是帮助产品推动人、项目干系人，以及团队成员理解什么是真正的价值。

相反，更好的做法是与开发人员和项目干系人坐在一起，考虑要做的事情。将所有人都认为最有价值的事情作为接下去要做的事情。这样做的真正意义在于学习怎样

达成共识。

接着，实现选定的功能特性，并尽快交付，然后倾听用户的意见，再重复上述过程。

第12章 是的，软件开发很难！



本书的早期读者曾这样对我说：“罗恩，你的阐述让软件开发变得清晰、简单、有吸引力。我们所有的工作都应该以价值为中心，根据价值去做计划，根据价值去管理，最后根据价值去实现功能特性。这些都阐述得很好、很到位。但是软件开发工作很难，可以说是困难重重！”

软件开发工作当然很难。这是由它的性质决定的，因为我们一直在努力突破自己。然而，本书的观点则是：对于做这样困难的事情，我们的应对方法就是坚持，直到

软件可以真正运行。要达到这样的目标，需要完成以下事情：

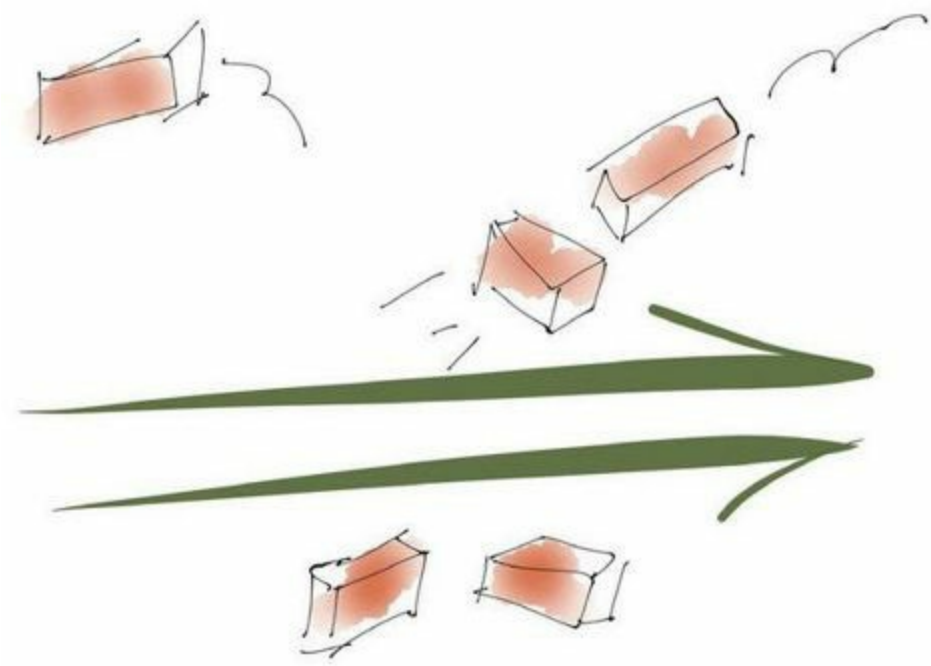
- 专注于我们看重的东西，这样才能得到最好的结果；
- 不时地开发出真正的软件，这样我们才能知道到底想要什么；
- 逐步构建我们想要的产品，这样才能知道实际做得怎么样；
- 学习所需要的计划、管理以及技术方面的能力，这样才能又快又好地构建产品。

软件开发工作的每个方面都很复杂，但是做这项工作并不需要很复杂的流程。在需要了解情况并进行相应的调整和完善时，太复杂的流程会迫使我们采取机械的做法，由此将我们禁锢住。

我们的确需要提高相关能力。通过一个旨在交付价值的简单流程，我们可以指导这一任务完成得更好。

最终决定具体“如何”做这些事情的人还是你，本书的重点则是帮助你提高相应的判断能力。并不存在一成不变的“最佳”方法，因此即使你只是决定暂时去做某件特定的事情，也需要检验它是否真的对你有所帮助。改变是不可避免的。

对于每一个想法，都暂且将它当作一种很好的做事方法。然后，把这种方法变成自己的，并由此发展出自己的想法。记住，一定要保持方法的简单！



可是，罗恩，这样还是很难！

的确如此，但情况会好转。

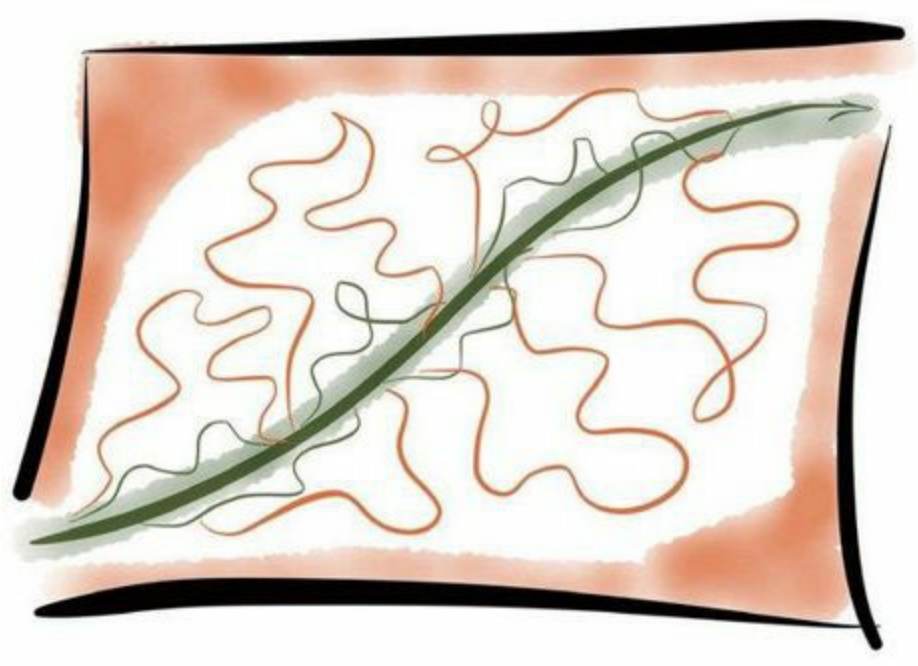
采用上述工作方式，我们就会提高得很快。作为团队，我们反复地决定做什么，然后完成它并对结果进行检验。虽然遇到的问题总是很难，而且构建优秀的软件总是很难，但是由于整个团队变得越来越强大，因此进展也会越来越快。

在接受膝关节置换手术之后（这件事情说来话长），我最开始几乎不能走路。只要走上几米就会令我感到疼痛，即使过一段时间后疼痛就消失了，我还是不能够走很远。但是我一直不停地尝试、不断地行走。慢慢地，痛苦减轻了，我也能够走得更远了。如今，虽然走得太远仍然会使我感觉疲倦，但是走几步路已经不再是问题了。

周期性的开发过程同样如此。一开始，我们会觉得很痛苦，同时进展缓慢。一段时间之后，我们就会完成更多工作，同时也不觉得那么痛苦了。可以说，这是一种励志的痛苦，它激发我们向上，并不是那种宣告我们破坏了某种东西的痛苦。

我希望能够带给你更好的消息，而事实是，要想优秀并不容易。不过，只要你愿意，并且不断地去做、不断地去改进工作方法，就能达到如你所愿的优秀程度。

第13章 事情并非那么简单



罗恩，事情并非那么简单。难道你认为我是傻瓜吗？

——转述自一位早期审读者

事情当然并非那么简单。然而，我们需要意识到，通常事情是可以接近简单的。当然，我们也希望它能够这样简单。

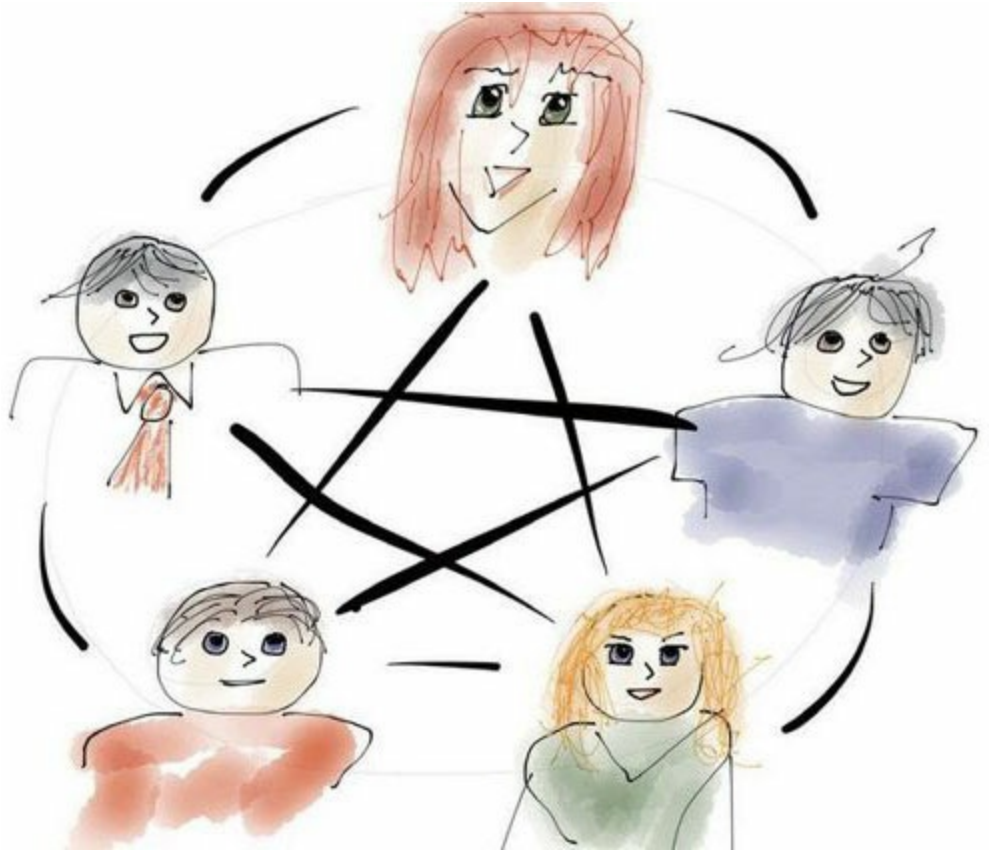
当走在各不相同的曲折小路上时，我们需要谨记：努力沿着简单的理想之路前行——专注于价值，根据价值管理和做计划，开发有价值的软件，并通过观察了解我们做得怎么样。

当我们真正去做时，事情可能会变得糟糕起来：实际业务中所有错综复杂的情况都

将出现，那些我们似乎已经放弃考虑的复杂情况也都将出现。

然而，所有这一切不外乎就是决定我们想要什么，从而引导我们获得它。实现途径则是构建软件并查看它的运行效果。

第14章 组建强大的团队



目的、自主与专精

不少人在做管理工作的时候，都觉得自己需要提供很多方向性的指导。然而，本书请求管理人员确保其团队成员知道要做什么，然后让他们自己去搞清楚怎样做。你也许要问，这怎么可能？

在《驱动力》一书中，作者丹尼尔·平克认为目的、自主与专精是提高员工满意度和工作效率的三大驱动力。对于他的想法，我深表赞同。它对探求软件开发的“自然

之路”很有启发意义。下面分别讨论这三个方面。

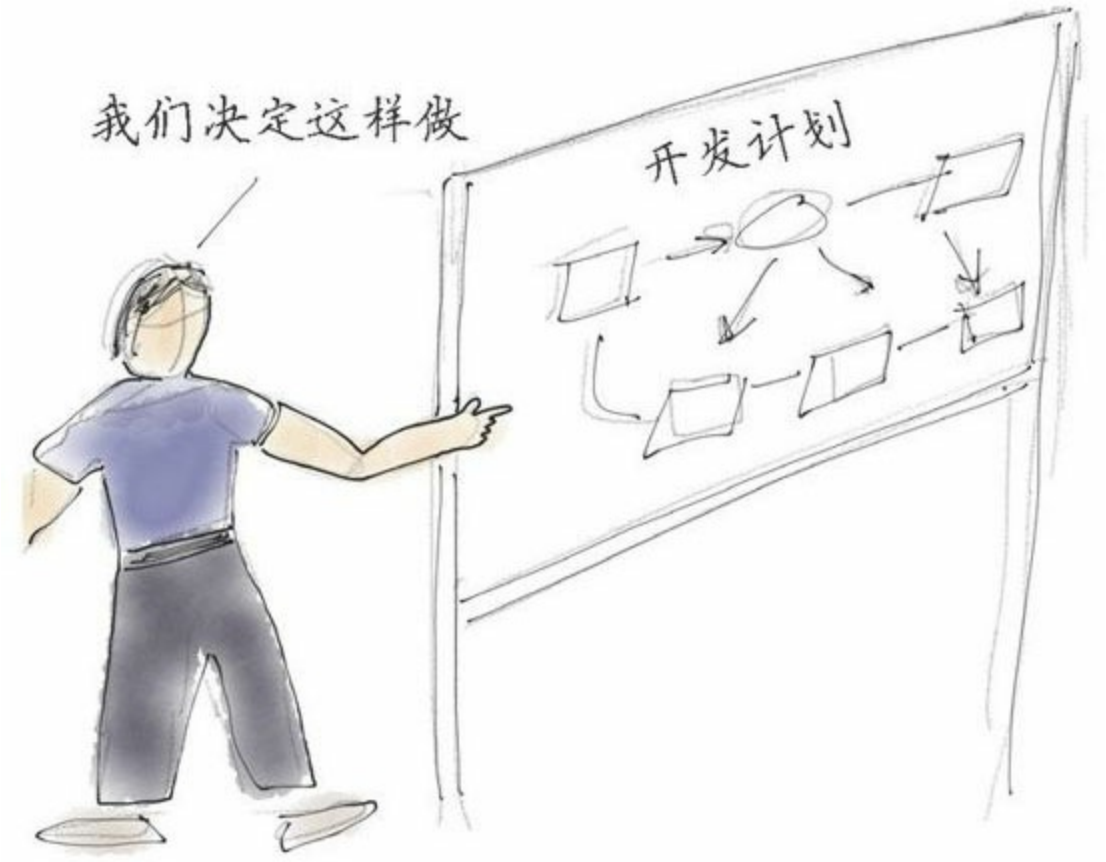
目的来源于具体的业务。

开发团队需要专门配备一位业务人员，由他来指导团队确定哪些功能特性需要首先完成，哪些可以推迟。这位专门的人员有时被称为产品负责人（Product Owner）或者客户（Customer）。本书统一使用产品推动人（Product Champion）这一名称，因为只有当整个团队都能够发挥主人翁精神做自己的产品，同时又有专门的业务人员提供愿景并“推动”产品时，才会出现最好的结果。

产品推动人为团队提供目的（不仅有大方向上的，而且还包括细节上的），并且保持每天都与团队接触，以确保团队理解为什么存在这样艰难的任务、最重要的问题有哪些，以及产品怎样能够最好地解决这些问题。产品推动人会将疑虑或者问题告诉团队，并让整个团队一起努力解决问题。

有一些团队的产品推动人会直接给出明确的解决方法，而不是将疑虑或者问题告诉团队。虽然这样做或许同样能够解决问题，但是这一做法并不值得提倡。由于是被“填鸭式”地直接告知解决方法，而不是通过主动思考，因此整个团队将需要更长的时间获得明确的目标感。

而当整个团队一起合作解决问题时，产品推动人能够更好地知道他到底想要什么以及怎样表达得更清楚。这样一来，整个团队以及产品推动人的能力都会有所提高。这就是目的带来的好处。



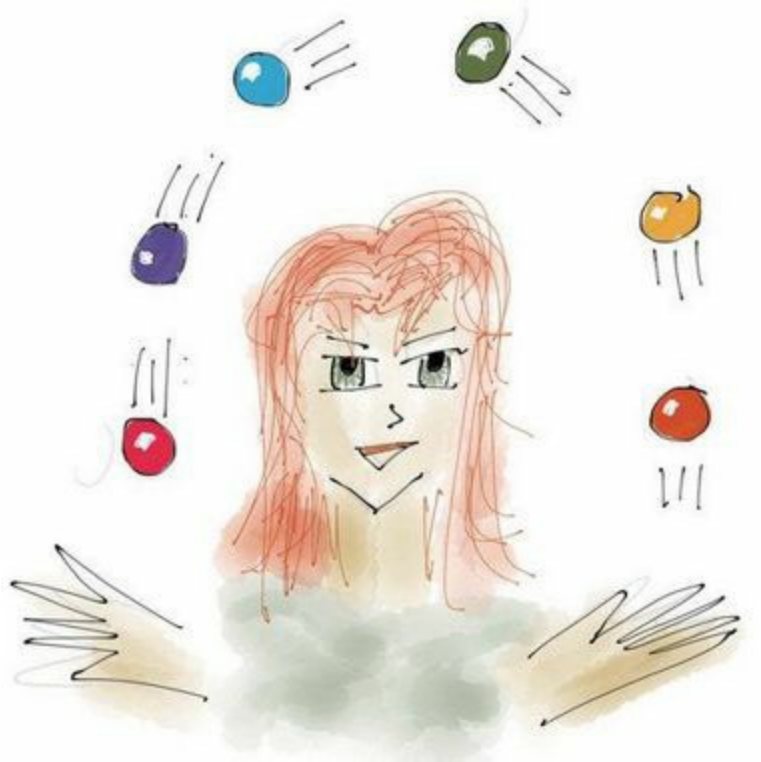
自主能够给整个团队带来责任感。

产品推动人负责说明待解决的问题，整个团队决定解决方法。在真正自主的情况下，团队并不需要任何人监督核查：他们自己做决定，然后自己开发，我们只需要查看最终的结果。每个人都能从这一过程中学习、受益。

在每一次为期两周的迭代中，团队都会与产品推动人讨论需要完成的功能特性。按照能够正常交付的标准确定工作量，并搞清楚怎样去做，再自主地完成这些任务。这样就能够交付真正的软件。

当团队能够自发地工作时，就会更加自主，同时也能激发更多的创造力去解决问题，工作效率因此也会更高。

自发的团队对于其工作目的会达成共识。这就是自主带来的好处。



专精源自迭代过程。

在每一次迭代中，团队都会在现有软件的基础上增加或者完善一些功能特性。一开始我们会遇到困难，但每经过一次迭代，当我们再次遇到困难时，就可以回顾之前的做法，然后决定怎样做结果会更好。这样，我们就能朝着专精迈进。

每经过一次迭代，我们都会完善“完成”一词的定义。所谓的“完成”其实就是一

种标准，达到这一标准就可以认为功能特性做得足够好。在整个团队朝着专精迈进时，我们也会使这一标准更加成熟、严格。

每一次迭代都是“完成”软件的一次实践。它以开发出真正能够运行的软件为目标，同时能够使大家都看到软件的质量如何，也有助于大家思考怎样使软件更好。同样，每一次迭代都是朝着专精迈进的过程。

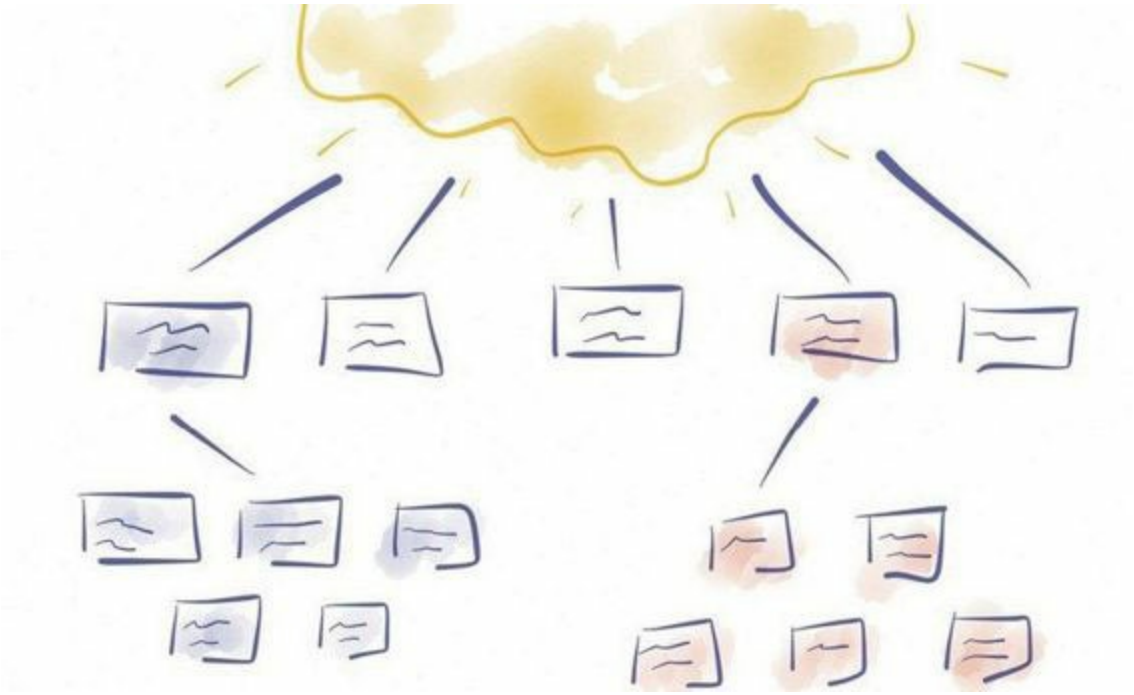
作为最流行的敏捷开发方法，Scrum的口号是“检视并适应”（Inspect and Adapt）。团队要观察哪些工作已经完成，同时注意哪些工作进展缓慢，并不断地改善局面。随着团队的不断进步，我们都在朝着专精迈进。

一言以蔽之：

在自主的专精团队中，每个人都十分清楚团队的目的。

这就是目的、自主与专精带来的好处。

第15章 使用五卡法进行初步的预测



如果不得不对较大规模的软件进行初步的预测，那么可以使用五卡法，这样既能够得到足够多的细节，又不至于陷入其中。任何宏伟的软件愿景都是由很多部分组成的，我们需要将这些组成部分切分为更小的模块。切分的标准是：团队成员认为自己能够把握整个模块并且能够在一周之内完成开发。以下是具体的切分方法。

- 第一步，列出三到五个最重要的“史诗级”组成部分，也就是你认为产品应该具有的大模块。用一句话来描述每个部分，然后将其分别写在一张卡片上。

- 第二步，将第一步中每张卡片上的“史诗级”模块细化到三至五张更小的卡片上，使每一张卡片上的内容更具体、更明确，当然范围也更小。确保其中的每个条目都具有商业价值，也就是说它必须是一个“功能特性”，而不能是某个技术想法或者部件。

- 最后，重复第二步，直到每张卡片上的功能特性的大小都相差不大。那么，多大才合适呢？团队成员认为能够在一周之内完成开发，这样的大小就是比较合适的。在划分功能特性的时候，还需要密切注意哪些功能特性的价值高，哪些价值低。暂

时将价值低的功能特性放在一边。记住，我们只是想确定大概的时间，并且通过认真细致的管理决定先做什么，而不是一味地督促员工（第17章将讨论这一问题）。

你的产品有哪些级别很高的功能特性？每一个这样的功能特性又有哪些细节？为了能够进行估算，你还需要知道哪些信息？

第16章 自然软件开发的管理之道

曾有人这样对我说道：“罗恩，在现实世界中存在不少管理人员。他们的存在自有其理由，你的模型却很少谈及他们。对于管理人员，你又有什么看法呢？”

我更愿意认为，我们的职责应该是创造“现实世界”，而不是去容忍它。管理的确是不可或缺的，然而我们需要的并不是通常意义上的管理。接下来就让我们进行具体的讨论。

当我们沿着“自然之路”进行软件开发时，大部分的管理工作是在团队内部完成的。在明确了产品愿景之后，产品推动人与项目干系人以及团队成员一起决定功能特性的优先级。因为团队每几周就需要演示一次目前完成的软件版本，所以每个成员都知道工作的实际进展情况。这既使得我们可以与团队之外的项目干系人进行相应的协调配合，同时也让我们很容易就能判断出团队是否需要某种帮助。

我们的跨职能团队拥有交付软件的每一个增量版本所需的全部技术与能力。团队成员能够自己进行测试，自己编写文档，所有与软件开发直接相关的事情都可以由他们自己去做。越接近这样的理想状态，所需的协调配合就越少。

同时，由于团队是自主的，因此团队成员会自主决定如何分配工作，并且保证所有工作都能够完成得很好。当整个团队能够以这种状态工作时，其实并不需要多少持续性的管理。

当然，有些管理工作仍然是不可缺少的。比如，虽然人员选择问题最好应由团队自己解决，但是人事决策问题，无论是招聘还是解雇，则都需要由管理层来决定。又比如，预算建议可以由产品团队与产品推动人共同提出，但是预算决策，无论是项目内部的还是跨项目的，则是一项管理职能。

不少管理者担心，如果放手这么多自己分内的工作，那么是否还能够实现自己的价值？让我们来思考一下这个问题。在管理学中，这通常被称为“授权”（delegation）。如果管理者能够组建一个高效的团队，而且该团队能够顺利地开发出其价值直观可见的产品，那么他能够做的事情不是减少了，反而是增加了。他可以继续去组建第二个、第三个这样的团队。

我们来思考一下彼得·德鲁克所提出的五大管理要素：计划、组织、人员配备、领导

和控制。我们将逐一考察这些要素，并从自然软件开发的角度解释这些思想对于管理者来说意味着什么。

罗恩，超越“通常意义上的管理”这个想法听起来很不错，但是我所在的公司是有长期目标的。我敢肯定，我们需要有长期计划。对此，你有什么好的建议吗？

管理层位于公司的金字塔尖，由他们来决定整个公司从事什么类型的业务。管理层从总体上决定要做什么、由谁来做。总体的计划活动是这项工作的起点，首先要确定待解决的问题以及可以把握的机会，然后再根据时间、人员配备和预算决定这些任务的规模大小。

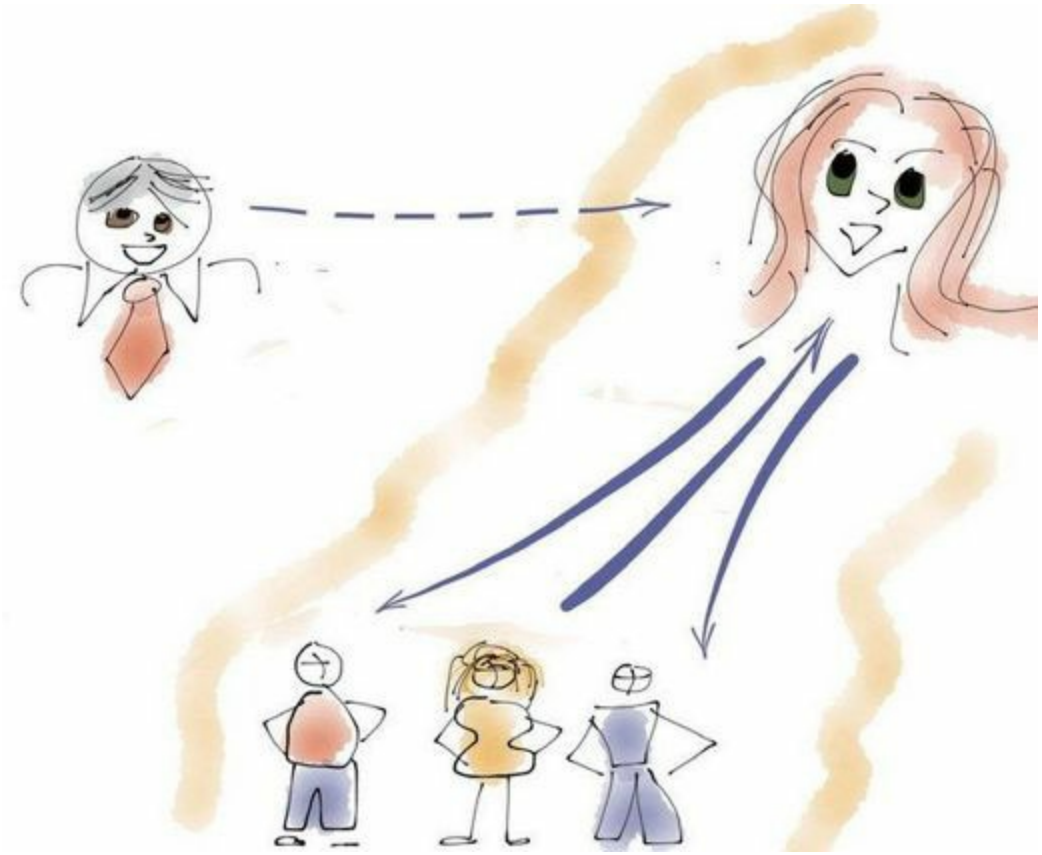
从自然软件开发的角度来看，用几乎任何方法都可以制订长期计划。所有职能领域的人都需要参与这一活动，包括管理人员、财务人员、产品人员以及技术人员。

如果按重要程度来排序，最重要的事情是什么呢？答案是，控制好自己所参与的产

品和项目的数量。先将手头的事情完成，然后再去做另外的。人的精力是有限的，同时做太多的事情只会使所有事情的进展都变得缓慢。

那么，做几个月或者一年的计划如何？

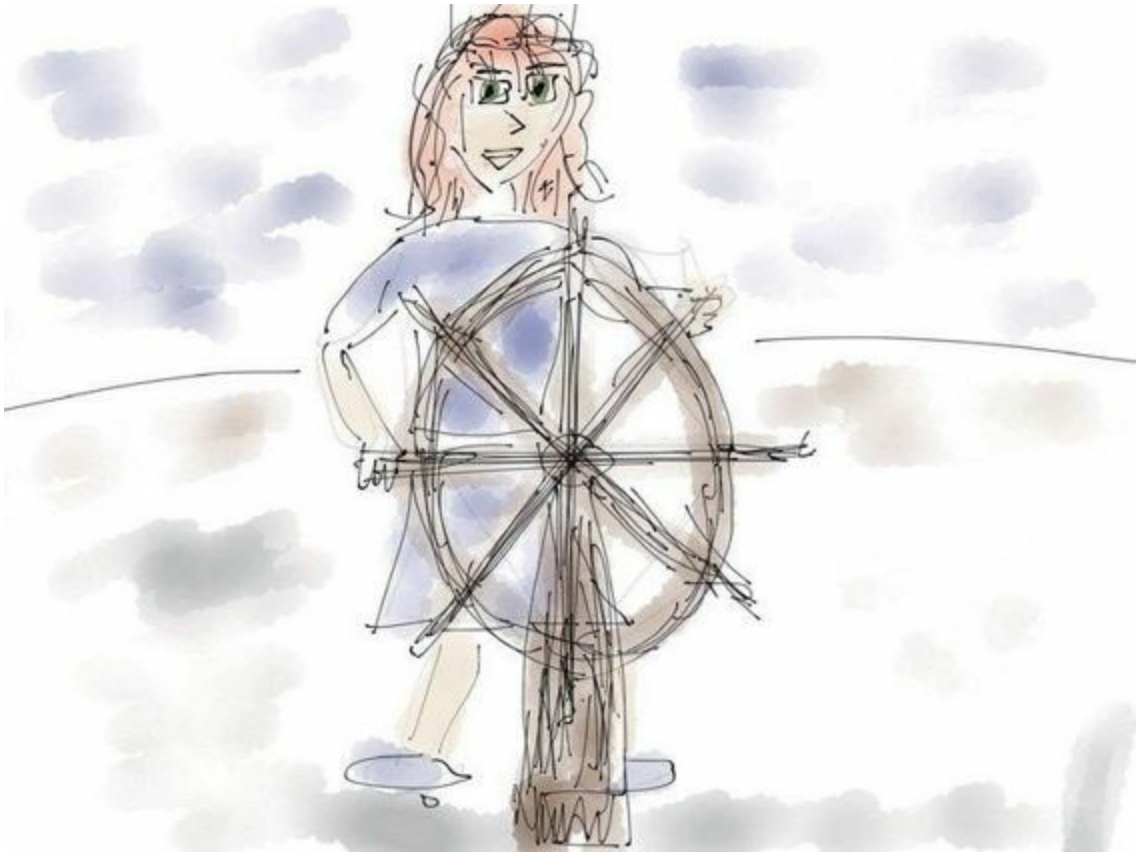
中期或者长期的计划通常以确定总体目标开始。按照自然软件开发的方法，我们首先构建高价值的功能特性，管理人员则可以随时关注我们的进度而不陷入具体的细节之中。高层的计划者和管理者需要十分清楚整个大项目需要有哪些功能特性，同时要求产品推动人通过演示实际可用的软件来展现这些功能特性。我们应该帮助产品推动人首先完成所有关键的功能特性，然后在时间和预算允许的条件下，继续完善那些较不重要的功能特性。



短期计划，比如说每天的或者每周的计划，又会怎么样呢？

采用自然软件开发的方法，短期计划持续不断地体现在开发层面上。功能特性越有价值，其优先级就越高。价值最高的功能特性会最早完成。时间一周一周地过去，价值也以可见的方式不断地增长。每隔几周，我们就会对计划进行调整，同时所有人都能够看到项目的进展情况。

这样做很简单。每隔几周，观察哪些任务完成了，并为接下来的几周做计划，同时确保朝着总体愿景和目标的方向努力。



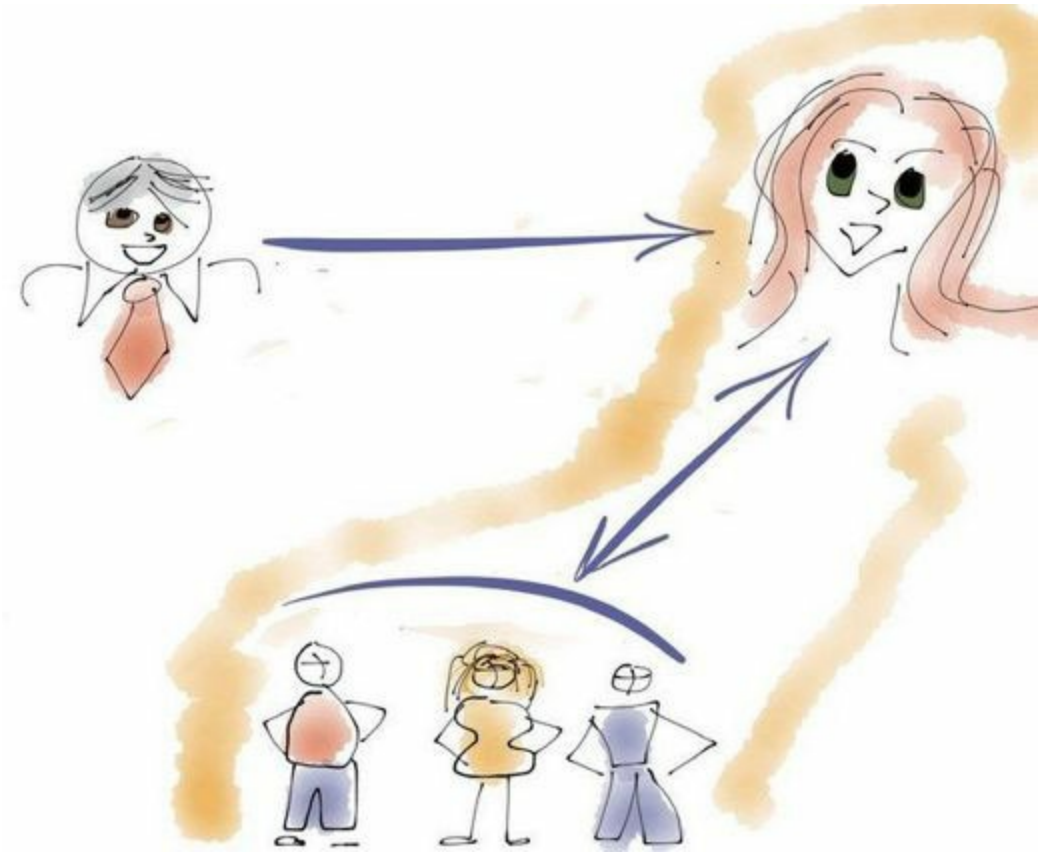
就我的经验来说，管理主要就是确保项目能够正常地进行。那么，如何确保不偏离计划呢？

老实说，我们的工作并不是死守计划。我们并没有固定的目标，一切都是为了能够获得最好的结果。

当我们经常部署产品并将真正的价值交付给用户时，会发现其实早在时间截止、经费用完之前，项目就可以完成了。为什么会发生这种情况呢？因为我们已经向客户提供了他们真正需要的所有功能特性了。同时，我们常常也会发现，无论在开发之前如何设想产品应该是什么样子，总是会有新的想法出现。采用自然软件开发的方法，我们就能够引导和控制整个项目，而不只是做计划并希望计划能够实现。

我们需要随时关注价值。在制订计划阶段，要不停地思考下一件最有价值的事情是什么。为了跟踪了解计划进行得怎么样，可以要求产品推动人演示软件，同时将产

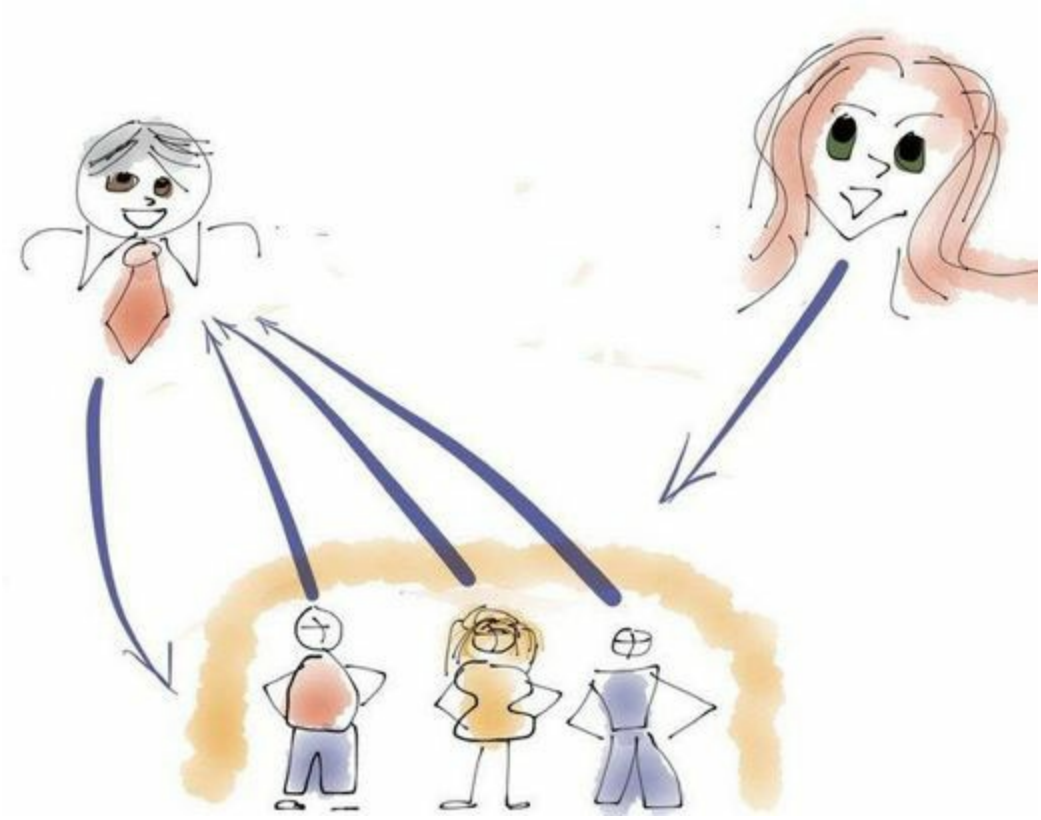
品的功能特性与我们所追求的价值关联起来。



罗恩，你给的建议听起来不错，但是我发现管理问题通常都可以归结为一件事：将合适的人放到合适的位置上。那么如何才能够最优地做出必要的组织决策呢？

高层的管理者同样需要担负起通常的组织职责，包括将资金与人员分配到具体的工作中。一般来说，在确定预算之后，管理层会选定产品推动人，同时很有可能也选定一些人事职员。通常，团队的核心人员由产品推动人选择，而剩下的人员则由整个团队来选择。在这一组织框架内，产品推动人与团队为了将工作做好而自发组织起来。

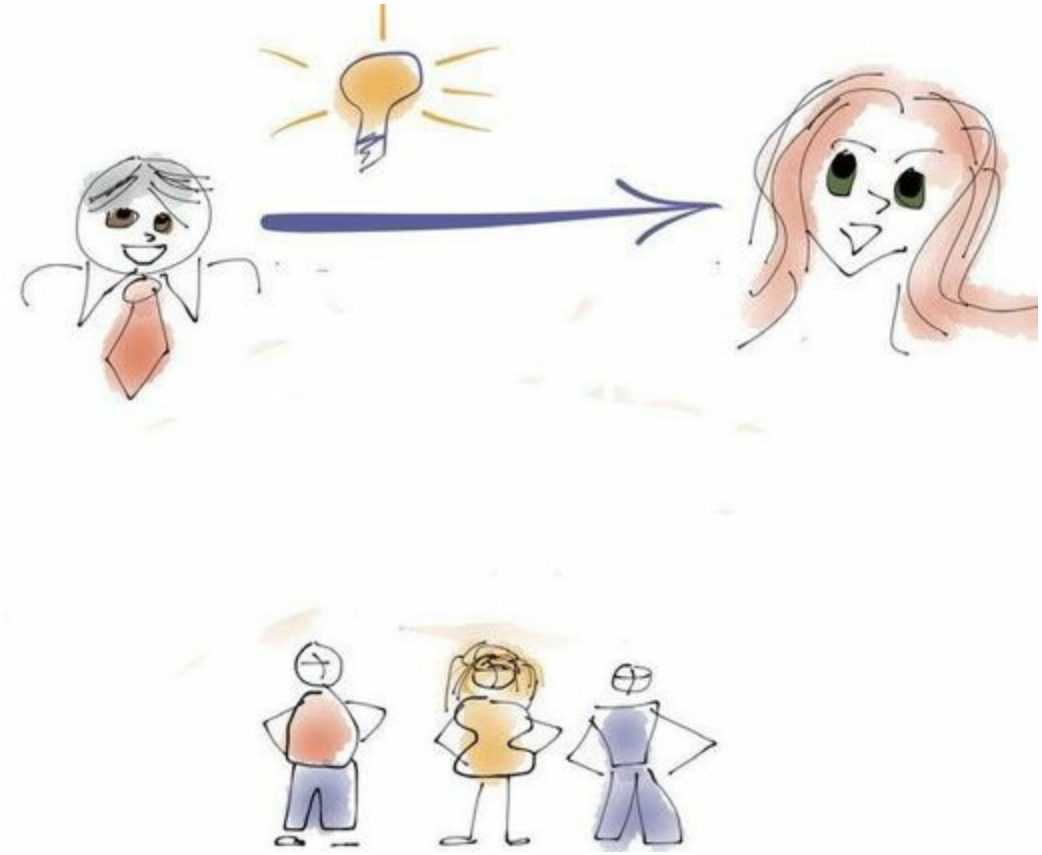
努力将比较小的组织决策权力下放到基层。同时，通过预算来控制任务规模的大小。尽最大的可能以结果为导向来评价工作的好坏。至于要做什么、怎样做，则交给最接近实际工作的人员负责。



那么，人事决策呢？由谁来作出招聘或者解雇的决定呢？

最有可能的情况是，人事安排必须由管理层提出，或者至少是得到管理层的同意。然而，越来越多的公司将决定权和建议权委托给团队。最好让团队决定他们是否需要增加人员以及增加谁，因为他们要比你更加了解团队需要什么。

除了帮助团队了解相关的政策、总体的招聘原则和策略之外，最重要的就是要了解现有团队成员的价值。团队对总体格局了解得越多，工作就会做得越好。



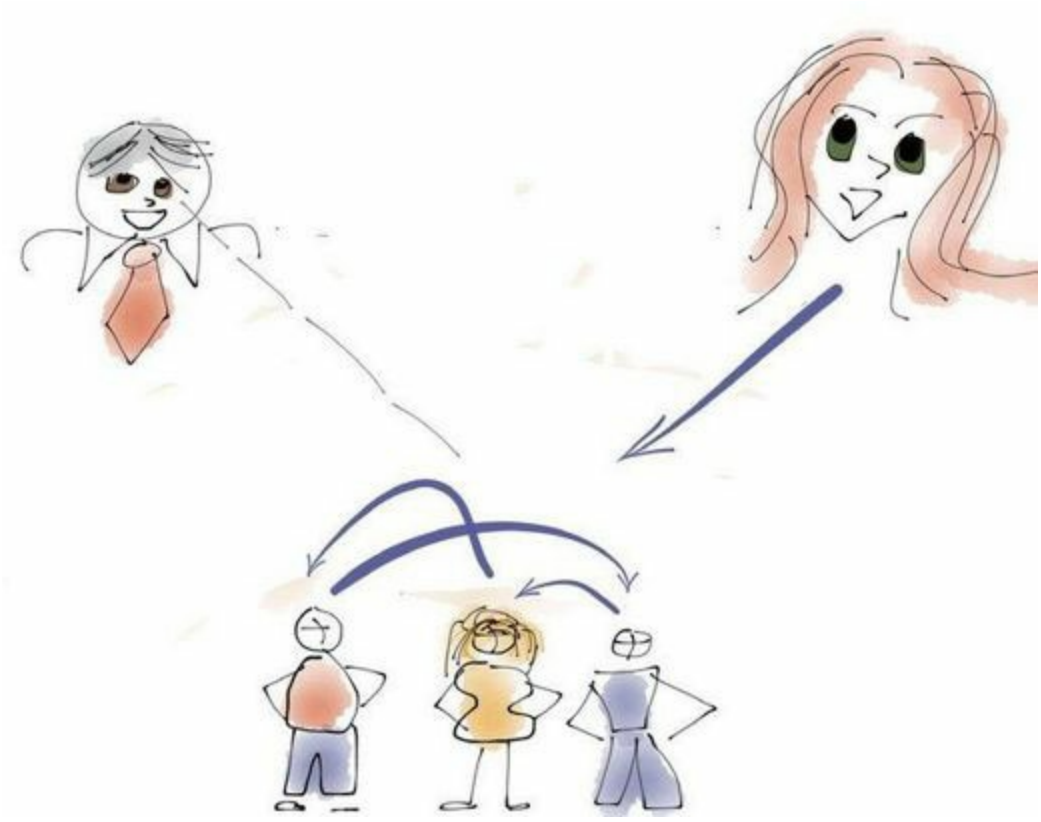
人们通常认为，管理就是对当前正在发生的事情加以指导。如果所有决定都由组织的底层作出，那么管理又怎样提供必要的指导呢？

基于长期计划，管理层就能够决定投资哪些产品和项目。之后，管理层会为每项工作选定一位产品推动人，由他来对这项工作的结果负责。在产品开发过程中，管理层会仔细察看，同时为产品推动人提供支持与指导，以确保产品与企业的宗旨保持一致。指导包含几种主要形式。

有时，情况会因为环境或者公司战略重点的不同而改变。这些变化会影响产品愿景、预算等方面。

有时，管理层会在察看工作成果时发现他们之前在某些方面表述得不完善，也可能会发现产品的某些地方提升了他们对用户需求的理解。所有这些都会造成团队愿景的调整。

有时，事情并不会进展得像往常那样顺利。如果按照本书推荐的方法工作，管理层就能够迅速地发现问题，并及时响应，帮助产品推动人及团队做得更好。当我们每几周就能够看见产品时，几乎不可能出现大的意外情况。



但是，工作总是需要控制的。我们如何能够保证每天的工作都在控制之中呢？

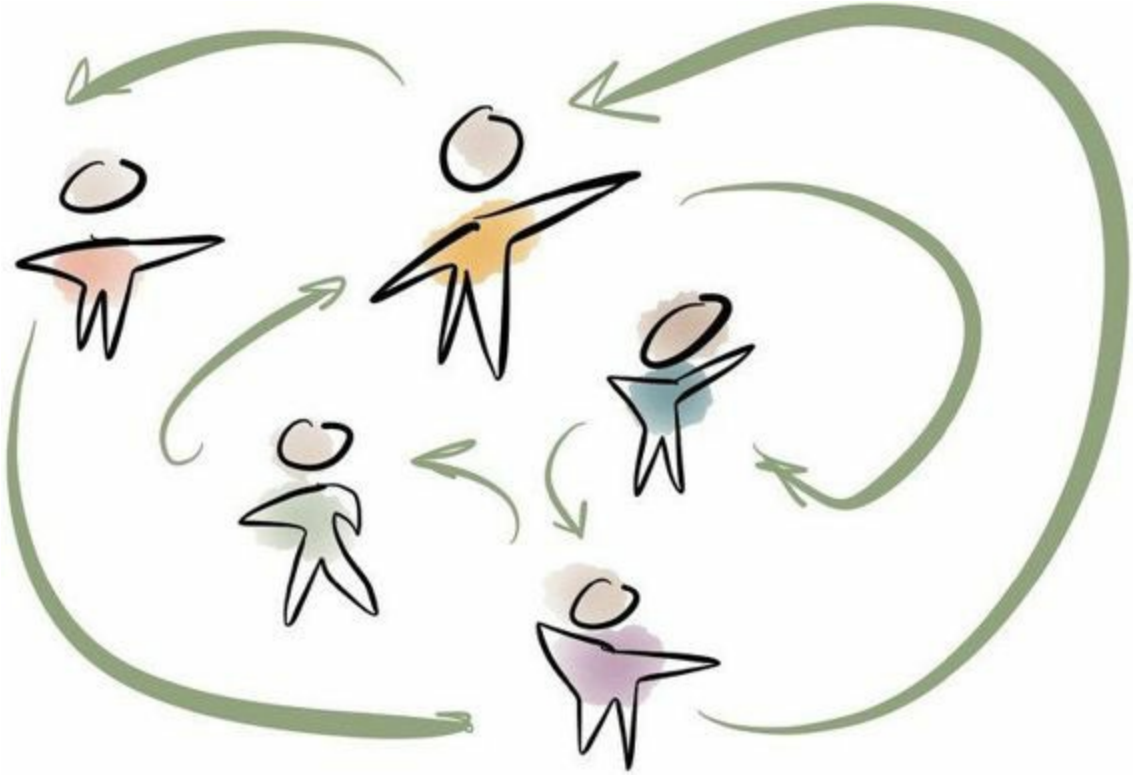
团队每天控制着自己的工作方式，产品推动人则每周控制着团队的工作内容。管理层关注工作的成果，同时确保工作的进度与已经花费的时间和经费相符。如果两者之间存在较大的偏差，管理层可能需要采取相应的行动。而所采取的行动，并不是干涉并参与具体的工作，而是向团队提供帮助和培训。当然，如果有需要，管理层也可能会调整预算、人员或者职责。

结论

自然软件开发的方法倡导将权力授予实际完成工作的人。这并不是什么新鲜的观

点，实际上管理从来就是这样做的。诚然，有一些管理人员由于担心在需要的时候提供不了帮助而对完全授权有些犹豫。幸运的是，自然软件开发的方法为我们了解现状提供了足够的透明度，这使授权变得很安全。

当开发团队向我们演示能够真正运行的软件时，我们总是能够知道他们的进展情况、他们在做什么以及做得如何。如果能够在遵循这些原则的同时结合自己的实际情况，那么你的工作肯定会做得不错。



第17章 监督员工更加努力地工作

曾有一位高级管理人员在被告知某个项目的实际进展速度要比预期的慢时，这样回答道：“看来我们只能监督员工更加努力地工作了。”

我发现很难从管理人员那里听到比这更令人反感的话了。而且，我敢肯定这样做只会适得其反。

当面对很大的压力时，开发团队会陷入各种错误之中，比如测试做得不够充分，或者对很糟糕的代码放任自流。这些既会减少当前工作产生的价值、延迟得到价值的时间，同时也会减少团队未来可以交付的价值。

当压力很大时，由于测试变得少了，因此软件中的缺陷也就变得多了。其中有一些缺陷难以被发现，从而会影响用户对软件的使用，并直接降低软件的价值。

也有一些缺陷会在产品发布前被发现，这通常意味着，在我们认为产品已经“完成”之后还会有一个测试阶段。显然，这会造成价值的延迟交付。

更糟糕的是，我们还必须花时间解决这些问题。这样的返工会使价值的交付进一步延迟，同时也使价值再打折扣。

最后，当团队在巨大的压力下工作时，代码的质量是很难保证的。糟糕的代码会使增加新的功能特性变得困难，而且会继续延迟价值的交付。

你的代码此刻有多少缺陷？难道你真的想让缺陷再多一些？开发人员说过多少次低质量的代码会减慢开发速度？你无意中造成的压力导致过这些情况的发生吗？

我们需要更多功能特性！为什么就不能加快开发速度呢？

坦率地说，当我去调查那些“需要更多功能特性”的公司时，几乎总会发现这些公司不会说“不”。他们已经变成了接单员，而不是决策者。或许他们正在做的事情会有一些价值，但实际上其中大部分工作并不会为产品或者客户带来多少真正的价值。

不，我们的确需要更多功能特性！开发人员必须加快速度！

事实上，你的开发人员已经在以尽可能快的速度工作了。当然，可能会有一些方法可以改善现有的代码，这样他们的开发速度就可以更快。同样，也有减少代码缺陷的方法，这样就可以缩短修复缺陷的时间。然而，这些优化工作都需要花时间，处于巨大压力之下的团队是不可能做到的。

一种在不知不觉中给团队造成压力的举动就是要求他们“加快速度”，这样一来就会出现上述问题。“加快速度”意味着要求团队“干完更多的活”。如此一来，他们就会尝试着这样做。而他们能够采取的方法，则无非是在质量和估算方面妥协。

一种可能是开发人员降低对质量的要求，那样就会带来更多缺陷，我们的速度反而变得更慢。显然，我们并不想要这样的结果。

还有一种可能：面对压力时，团队成员会开始有意识地或者无意识地对所承担的任务更加保守。与以前相比，他们会将任务评估得更大或者更难。这会让人觉得他们的开发速度更快，而实际情况并非如此。

那么，为了能够在单位时间内有更多的产出，团队都需要做些什么或者了解什么呢？你相信“更加努力地工作”会是这一问题的答案吗？管理层能够做哪些事情来帮助团队真正地提高工作效率？

有什么方法能够提高速度呢？我们怎样才能够做到？

如果一定要更快一些，分析导致延迟的各种原因。这通常要比提高个人的工作效率更有作用。

首先，我们优先要提高的是团队的工作效率，而不是个人的工作效率。既要确保每个团队都有很好的技能组合，还要保证团队有完成工作所需的全部技能。那些具备

关键技能的团队成员必须是全职的，不能同时参与多个项目。此外，还需要通过专家的指导来提高其他成员的能力。尽可能让所有的团队成员都在一起工作。

其次，通过提高个人能力来提高个人的工作效率，而不是依靠督促他们更加努力地工作。“更精明地工作，而不是更努力地工作”，这意味着我们要帮助员工变得更精明。

导致你的项目延迟的真正原因是什么？是决策造成了延迟？还是修复缺陷？抑或是任务在个人或者团队之间的移交造成的？你自己造成延迟的原因又是什么？请找出这些原因，然后解决问题。

那么，我们至少可以根据速度来预测项目的完成时间吧？

答案就一个字：不！最好的情况并不是我们正确预测项目的完成时间，而是当我们选择在某个时间完成时，能够确保在截止时间到来之前顺利地完成最有价值的功能特性。通过选择最重要的功能特性并每几周就构建出一个可发布的版本，自然软件开发的方法使我们能够做到这一点。

无论是采用估时、用户故事点数还是用户故事数量的方法，利用速度来预测完成时间几乎总是项目将会出现问题的表现。

本书假设的前提是：我们想要价值，想尽快获得价值，并且想优先获得最大的价值。正如上文所述，达到这一目标的最佳方法就是要确保总是“完成”工作。如果团队总是朝着公司的固定目标工作，那么极有可能会“做错”。这种做法很有可能导致管理基于成本而不是价值。这其实是一种很愚蠢的行为。

为提升技能提供真正的机会（详见第14章）。

真正能够帮助一个团队提高工作效率的是更高的技能。这意味着，在培训和教育方面的投入将以工作效率的形式得到回报。

人们在日常工作中已经够辛苦了。他们在有限的个人时间里利用晚上或者周末提升自己的能力，这种设想现实吗？我们怎样做才能够使这种设想更有可能成为现实？

大部分员工的日子过得都不会很富裕。设想他们会利用自己的闲暇时间花钱去学习课程或者参加昂贵的会议，这现实吗？我们怎样做才能够确保他们有钱、有时间去学习，而且这些钱和时间真的用在学习上？

请不要再“监督员工更加努力地工作”，帮助他们提高能力吧！

第18章 能力是提高速度的前提

我们可以通过简单的、较短的迭代构建完整却很小的产品，但当我们这样做时，很容易出现各种问题。这是很正常的。接下来，我们看看主要有哪些方面会出错，以及错误发生时如何应对。请认真思考以下建议所体现的态度，而不仅仅关注具体的内容。

开发团队反映说，他们不可能在两周之内将所有这些功能特性都做好。

是的，这的确是一个普遍存在的问题。通常在最开始的时候，开发团队并不能够在两周的时间里发布完整的产品增量版本，他们都会要求多增加一些时间。恰恰相反，我建议给他们更少的时间。譬如说，要求开发团队在一周内完成一个完整的产品增量版本。

开发团队很有可能会自己琢磨出解决方法。他们会明白其实只需要做很少的改变，

然后再将注意力放在这些改变的集成和测试上，最终使整个软件能够运行即可。

当然，也有极少数的情况是，团队成员会走过来跟我说：“只有一周，我们什么事情都完成不了。” 顺便说一句，这种情况出现在讨论中的次数比实际发生的次数要多。当听到这样的话，我通常会问他们：“哦，那就是说给你们一天的时间，你们也是什么都完成不了，是吗？” 他们都回答说是，然后我就会接着问，如果什么事情都不准备去做，那么为什么明天还要来工作呢？哈哈，很有趣吧！但我是认真的。

通常，这样的提问都会使随后的话题转向为什么完成一次增量式开发会如此困难。一旦我们开始询问为什么，就很容易弄明白要去做的事情。

这些问题的答案通常都会归结到开发时间、集成时间和测试时间上。

如果问题仍然不清晰，那么就要求开发团队重新生成一次系统，并在不做任何改变的情况下使系统随时可以部署。如果这样的任务也不能在一周之内完成，就能够确定问题到底出在何处了，不是吗？

无论如何，软件的构建过程都是比较慢的，而且是不太可靠的。好在我们都是程序员，可以解决这个问题。

产品推动人和管理者比开发团队更想要解决这一问题。他们需要定期看到已经完成的产品增量版本。每一次增量式开发的完成都有着很大的价值，这就是为什么要求添加独立的或很小的功能特性，甚至不要求添加任何功能特性是很正确的商业决定。千万不能要求开发团队关注某个技术问题！应该要求他们呈现可以运行的软件。

开发团队又说，在经过测试之前，他们不能够发布任何版本。

作为团队的产品推动人或者产品经理，你知道这是不能接受的。当质量保证工作发现软件的缺陷时，由于与之相关的修复和测试工作需要重新进行一遍，因此会使你不能够将全部精力都放在商业目标上，从而会干扰商业计划。

而这一问题的解决方法既很直接，又很困难：当将软件交付给质量保证团队时，必须确定它不会因为其中包含缺陷而被退回。

具体的做法就是自己进行测试。验收测试驱动开发（ATDD）和测试驱动开发（TDD）这样的技术实践能够使我们很好地做到这一点。采用这样的技术实践之后，我们在交付软件时就会越来越有信心，知道软件将会像预期的那样工作。要做到这一点，团队成员必须精通验收测试驱动开发和测试驱动开发。

有时候，我们对外部的依赖并不在测试方面，而是某些其他“资源”，如数据库管理部门或者用户界面设计部门。这一问题的解决方法是组建跨职能团队，也就是说要将所有这些技能都引入自己的团队。最好的做法是，对于重要的项目，将这些部门的人员分配到项目团队中来，使他们真正地成为团队的一员。

但是开发团队接着说，真正开发完成软件仍然需要一段时间。

这很正常，对于“第一次”，我们愿意花时间来完成这一目标。学习构建可靠的产品增量版本的确需要时间，特别是当团队正在艰难地面对大量用老式方法编写的代码时。好在这样的困难并非无法解决。敏捷方法Scrum就有这样一个概念可以帮助我们：不断演变对“完成”的定义。

我们要求团队在每一轮冲刺（sprint）中都完成一个产品增量版本。这一增量版本

必须满足当前我们所谓的“完成”标准。而且，随着时间的推移，“完成”的定义将变得越来越严格。

因此，在最初的迭代中，我们甚至都不要求对增加的功能特性进行集成。验收时，我们在开发人员的机器上逐一察看功能特性。这样的验收方式会使业务人员感到很担忧：怎么能知道这些功能特性可以协作呢？因此，我们将改进“完成”的定义：所有的功能特性必须能够在同一台开发机器上运行。为了达到这一要求，开发团队在短时间内所承担的功能特性可能会变少，但是由于我们的要求是呈现能够真正运行的软件，因此这是不错的商业决定。

接下来我们可能会发现，虽然所有的功能特性都集成了，但并不能很好地协作。我们意识到需要进行更多的测试，因此会在“完成”的定义中增加一些测试标准。

这又会使我们发现，必须在迭代的一开始就知道最终的软件要通过怎样的测试。这将促使我们能够更好地从产品推动人那里了解实际的使用场景，同时也帮助我们更好地针对到底需要什么达成共识。

就这样，通过持续强调“演示真正能够运行的软件”，开发团队的工作流程得以改进。在做了一定的工作之后，他们就能够真正地完成软件。

现在所有人都会说，似乎这样做太慢了！

刚开始，的确会感觉这样做很慢。但是到后来，就会觉得顺利、高效、不慌不乱。每几周就能够有一个可以发布的、已完成的产品增量版本，我们绝对想要这样的结果。为了达成这一目标，软件必须经过充分的设计和测试，而且每个方面的工作都是全面的。这也就意味着，开发团队在每次迭代中能够承担的工作比以前的要少。因此，感觉慢很正常。而与以前的区别是，我们能够真正地完成工作。

既然越来越接近于真正地完成工作，速度明显变慢就是一种错觉。我们减少了很大一部分过去存在的不正常情况的发生，即在应该已经“完成”之后还持续出现的漫长的、令人痛苦的测试与修复过程。难道你都忘了在应用程序应该发布之时或者发布之后你熬过的那些时光吗？

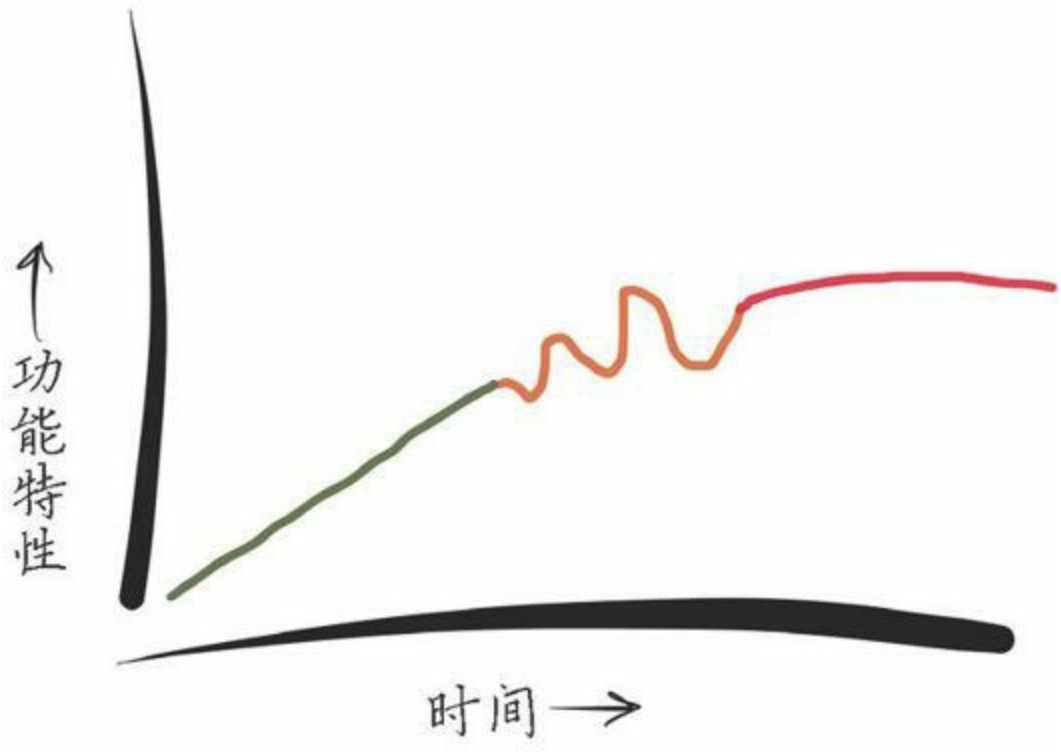
此外，因为我们是在进行增量式开发，即优先做价值最大的事情，所以能够在截止时间之前交付最有用的价值。然而前提是，所交付的软件必须可用。

的确，在最初的一段时间里，我们会觉得这样做很慢。但是即使在这段时间里，这样做也不可能比以前更慢，因为缺陷减少了，而且更整洁的代码能够使工作更加顺畅。因此，就这样做吧。随着团队越来越熟悉这样的做法，他们就能够在迭代中承担越来越多的功能特性，情况就会变得越来越好。

结论

为了加快开发速度，我们能做的最有价值的事情就是提高团队成员的技能。这一投入很快就能带来以下回报：浪费在修复缺陷上的时间会更少、开发过程会更加顺畅。不要将迅猛当作高效。速度最快的团队总是平稳、优雅地前进。

第19章 重构

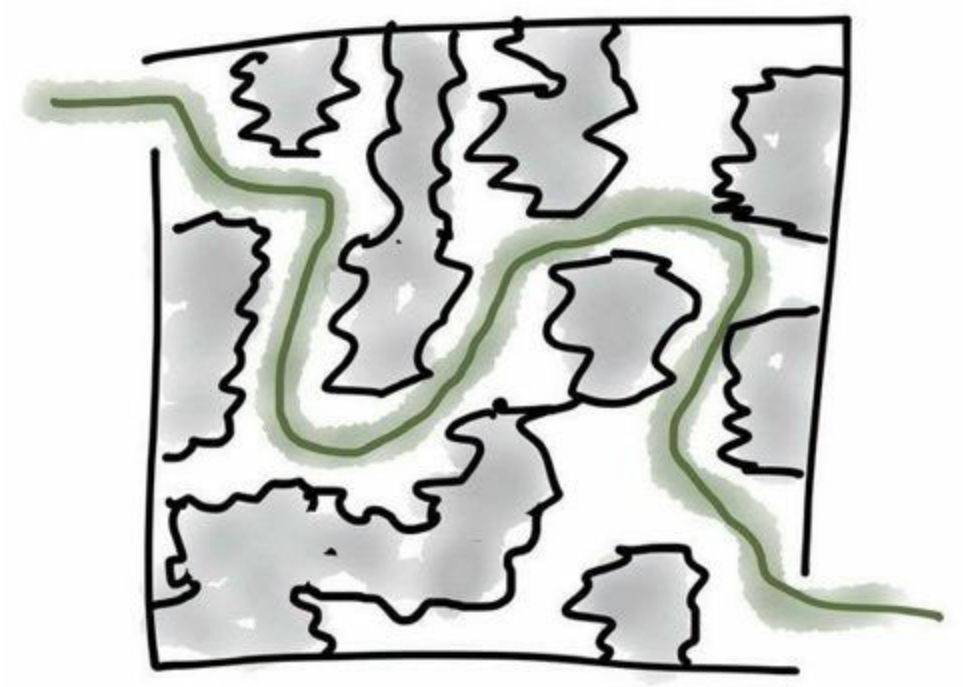


温馨提示：我们需要稳步前进。为此，需要时刻保持设计的清晰和整洁。而为了做到这一点，则必须进行重构。

沿着“自然之路”进行软件开发，要求我们以可见且可用的软件展现平稳的进度。看到真正的软件以稳定的速度在形成，我们便可以知道当前所在的位置。这帮助我们决定接下来做什么，以及哪些工作可以推迟。

然而，我们的速度常常并不稳定。即使工作内容的难度看起来都差不多，我们的速度也有可能变得不确定。一旦这种情况发生，速度几乎将不可避免地慢下来。

这会挫伤团队的积极性。更糟糕的是，它会使做计划变得很困难。同样糟糕的是，缺陷不断增加。最糟糕的是，我们可能不能够在截止时间之前得到尽可能好的产品。

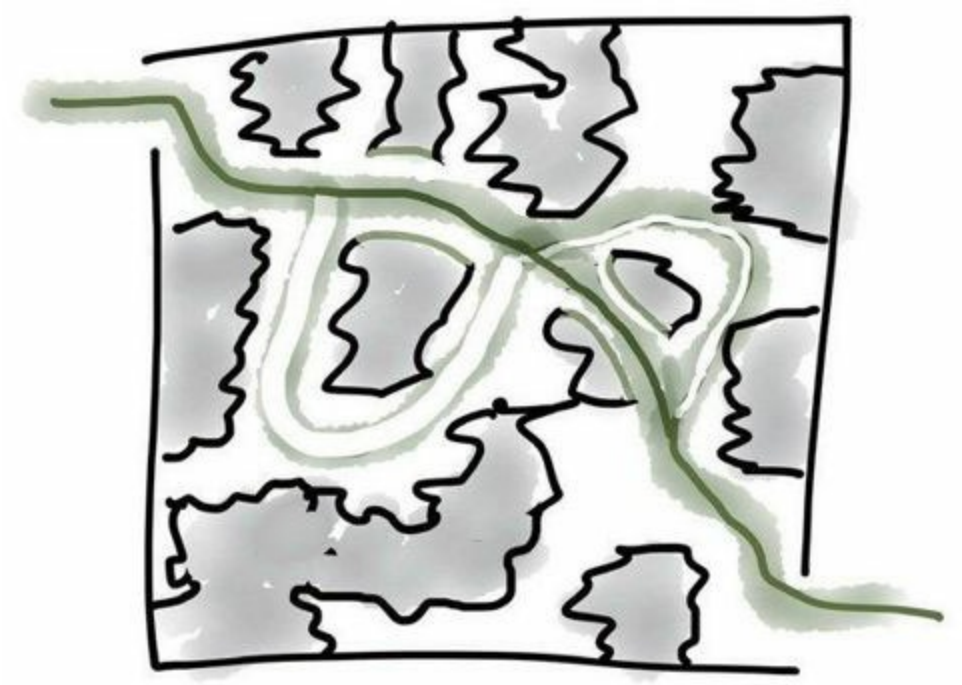


曲折通道

构建一个功能特性所需要的时间大致来自以下两个主要方面：一个是它本身固有的难度，另外一个则是将它加入现有代码时可能的难度。对于功能特性固有的构建难度，开发团队一般会估计得比较准确。因此，使开发速度变得不确定或者慢下来的就是将功能特性加入现有代码时可能的难度。我们称这一难度为“劣质代码”。

如果我们允许代码的质量下降，那么有些功能特性可能很容易就被加了进来，而另外一些看起来相似的功能特性则可能陷入劣质代码所形成的曲折通道中。这导致两个看起来相似的任务所需的时间大不相同。

为了保持进度的平稳，我们必须避免这种曲折通道；而当这样的通道确实存在时，我们需要做的就是使它变直。

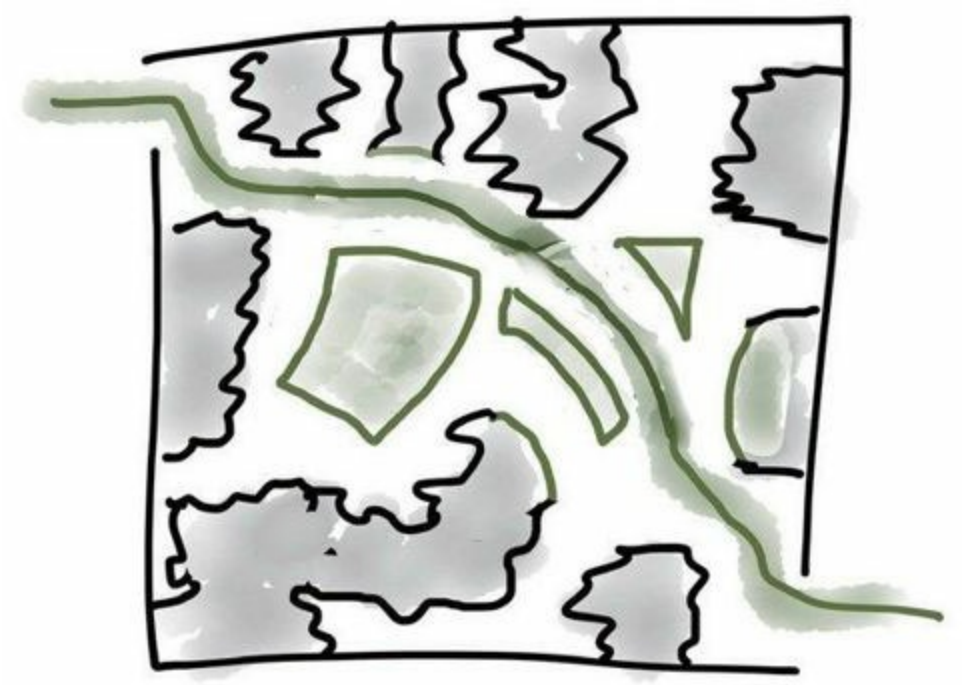


重构能够使通道变直。

重构一词指的是保持代码整洁的一种简单的、经常性的过程。通过重构，我们努力避免构造那些会使开发速度变慢的曲折通道。当曲折通道真的出现时，应该将它变直。

让开发团队不愿意花时间重构很容易，只需要多给他们一些压力即可，比如要求他们完成更多的功能特性。但当我们这样做时，开发团队就会求助于程序员的“秘诀”——不充分测试和编码捷径。这所造成的后果是，软件会有更多缺陷，同时进度也变得不确定，最终导致开发速度变慢。

工作的好坏很难一眼看出，至少从表面发现不了：它隐藏在表象之下。然而作为商业人士，我们当然需要好的工作成果。因此，我们必须期望并要求开发团队保持代码的整洁。如果我们觉察进度变得不确定或者速度变慢了，很可能就是应该重构代码的时候了。这也意味着，是时候减轻开发团队的压力了。



当所有的通道都变得非常曲折时，怎么办？

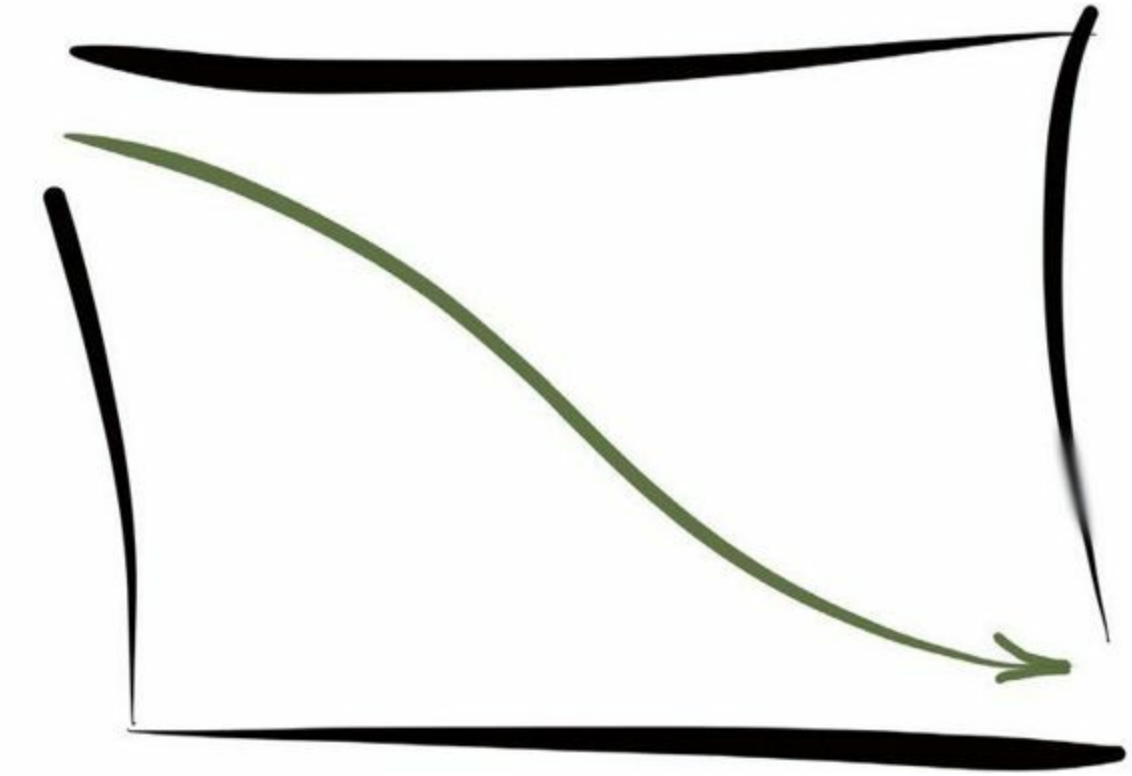
通常来说，将现有产品中某个较大的部分推倒重来并不是很好的主意。可能有些人认为这个主意不错，但是我敢肯定它并不是。

同样，完全停止功能特性的开发来“清理”代码也不是一个好主意。可能也有一些人认为这个主意不错，但我仍然敢肯定它不是。

看来遵循露营地规则（Campground Rule）似乎是最好的选择：在离开露营地时，要让它比你来的时候更好。每次要构建一个功能特性时，我们都先去清理那些将要用到的代码。并不需要让这些代码变得完美，只要能使我们比较容易地加入新的功能特性就足够了。这样，一旦功能特性可用，我们就清理应该清理的代码，外加一点相关的代码。

这样的过程正是我们需要的：在相关的地方做少量的代码清理工作并不会使速度变慢，因为大部分代码仍然保持不变。而我们投入大量工作的那些代码由于受到更多的关注，自然很快就变得整洁了。

对于开发人员来说，重新编码总是很具有诱惑力，但这种做法几乎从来都不是最佳方案。相反，掌握重构技能并学会应用露营地规则才是最佳选择。



第20章 敏捷方法

我很幸运能够成为《敏捷宣言》的起草人之一。在本书中，我已尽最大努力使书中内容与《敏捷宣言》保持一致。可以说，“自然之路”是我从事软件开发半个世纪并且实践敏捷方法将近20年以来所学知识的总结和提炼。我无意再创造新的敏捷方法，而只想根据我在敏捷概念诞生之前、之时以及之后所学到的一切，来阐述我认为正确的软件构建方法。

若想了解关于敏捷软件开发的更多知识，有很多敏捷方法或框架可供学习。其中最流行的当然是Scrum，它由Jeff Sutherland和Ken Schwaber提出。Scrum并非只为软件开发而设计。因此，它并没有明确地包含验收测试驱动开发、测试驱动开发、重构等技术实践。我想说的是，为了让使用Scrum的项目取得更好的结果，需要

加入这些技术。

由Kent Beck提出的极限编程（Extreme Programming，XP）则是一种敏捷框架，它明确地包含了以上技术。除此之外，极限编程与Scrum十分接近。虽然极限编程并没有Scrum特有的“Scrum专家”这一角色，但它也经常建议要有一位教练，这其实是一个相似的概念。我的理解是，Scrum结合上述技术实践就是极限编程。当然，有些人并不赞同我的观点。

Alistair Cockburn的水晶方法（Crystal Clear）则是比Scrum更简单的一种敏捷框架。还有一些复杂的大规模类敏捷框架，如动态系统开发方法（Dynamic Systems Development Method，DSDM）、Craig Larman和Bas Vodde的大规模Scrum（Large Scale Scrum，LeSS）、Scott Ambler的规范敏捷开发（Disciplined Agile Development，DAD），以及Dean Leffingwell的规模化敏捷框架（Scaled Agile Framework，SAFe）。还有很多，在此不再一一列举。若感兴趣，可以去读一读。

我想再次说明，本书并没有提出新的敏捷框架。相反，我想请你思考：要使软件项目进行下去，特别是那些将采用敏捷方法的软件项目，都需要哪些条件？这样，无论采用哪种框架，你都能够取得成功。

然而，关于框架，我的确有如下建议。

- 框架并非越多越好。正如《敏捷宣言》所述，我们认为“个体和交互胜过流程和工具”。对于项目来说，框架应该更像是运动服而不是紧身衣，它在大致合体的基础上更注重的是宽松。每一个参与项目的人都需要有“自由活动”的空间，以便用任何框架都没有考虑到的、也没有被任何规则所支配的方式进行交互。

- 框架要尽量轻。当然，有很多团队参与的大项目所需的流程肯定比只有同在一个办公室里的六人团队参与的项目要多。然而，即使是这样，也要保持所用的框架尽可能轻。可以利用回顾会来决定增加哪些流程。将增加流程元素当作实验，同时想清楚要从流程的改变中得到什么，并根据结果去检验。如果没有得到想要的结果或者得到某些不想要的结果，那么下一次就尝试另一种改变。

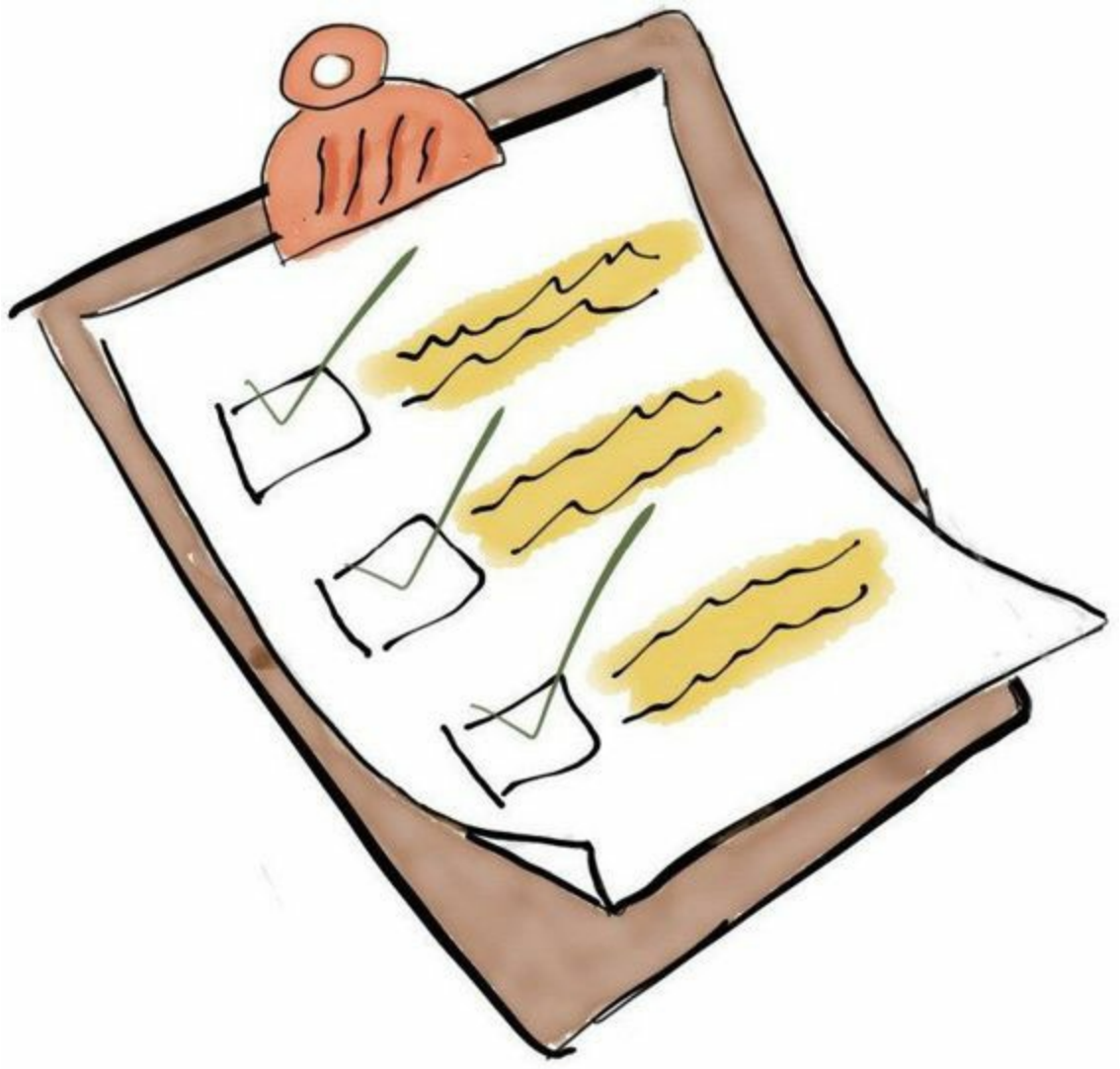
- 掌控框架，而不要被框架掌控。对框架进行调整，以使项目更有效率，但也不能仅仅因为框架要求做的事情很困难就去改变它。本书中的一些想法，以及你所用框架背后的一些思想都是帮助你进步的挑战。根据自己的能力进行调整，但挑战自己是必要的。

- 保持流程的改变贴近团队的实际。大范围强制推广一些做法并没有什么好处。通过检验实际运行的软件来管理项目，从而使软件的功能特性逐渐增加并且一次次地顺利通过业务验收测试。同时，通过检验团队的工作成果以及产出速度判断工作情况。不必要求团队一定遵循某个特定的流程，这样做可能会适得其反。

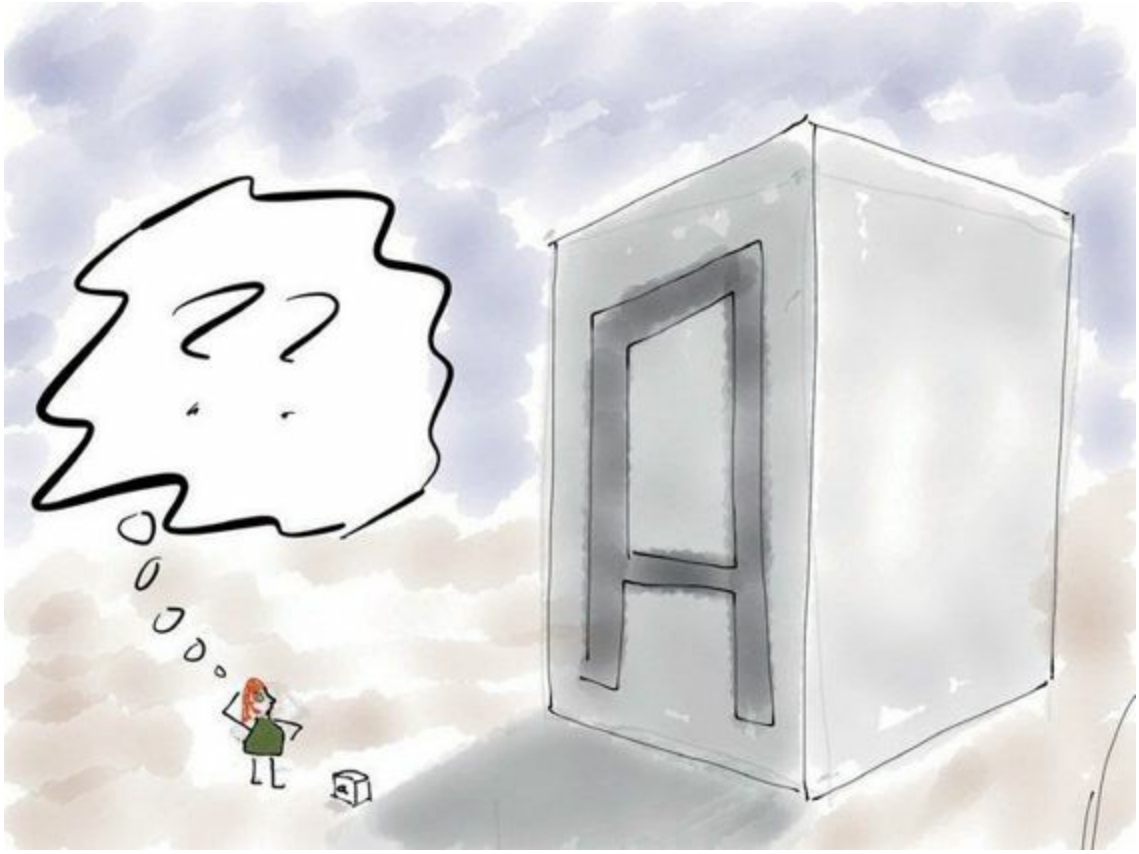
- 把学习放在首要的位置。正如本书所述，从最高级的业务高管到管理人员，再到基层的技术人员，所有参与项目的人员都需要具备技能。特别是每天都与软件打交道的技术人员，他们需要具备特定的技能才能把软件做好。我希望你能够在项目培训和支持方面有一定的投入，这将为你带来更好的软件以及更快的交付速度。

- 还有最重要的一点：思考。构建有价值的产品是很复杂的工作。要做好这件事，并不能依靠参与并控制一切，而是要依靠观察所发生的事情并随时作出反应。这有点像团队比赛：可能会有一些方案，甚至还可能有一些预先计划的行动；然而，比赛时所发生的情况总是会与预计的不同，而胜利则取决于团队成员的现场配合能力。矛盾的是，这样的能力其实来自于采取行动之前所做的思考。再重复一遍，思

考！



第21章 大规模敏捷



你需要的不是大规模敏捷，而仅是敏捷

最近有很多人都对“大规模”敏捷很感兴趣，它已经成为了一件大事。大公司都听到了敏捷术语所吹响的号角。正如过去听到诸如六西格玛和全面质量管理这样一些优秀的思想时所表现的那样，大公司现在都想要“敏捷”起来。敏捷已经成为它们要做的事情。然而，它们都是大公司，因此也就理所当然地认为自己需要的是大规模敏捷。

事实证明，在大多数情况下，它们都错了。大公司需要的并不是大规模敏捷，而仅是普通、简单的敏捷软件开发技术。

大规模敏捷对于供应商来说是好事，但这并不意味着对于你来说也是如此。

大规模敏捷目前已经成为值得参与的事，因为人们都觉得自己需要它。既然已经形成了很可观的市场规模，在大规模敏捷中出现相互竞争的方法也就再正常不过了。如今随着大敏捷市场的不断增长，肯定会涌现出越来越多的方法。

至于目前都有哪些方法，我想还是让你自己去寻找并从中做出选择吧。我想指出的是，除了一种可能的例外情况之外，这些方法都会使你误入歧途。

然而，这并不是说大规模敏捷不会成功。它很有可能会成功，只不过这里所说的成功是针对咨询和培训公司而言的：大公司会购买大规模敏捷的产品并引入相关的概念，使相关的咨询和培训公司赚得盆满钵满。

滚石乐队有一首歌这样提醒我们：你并不能够总是得到你想要的东西。

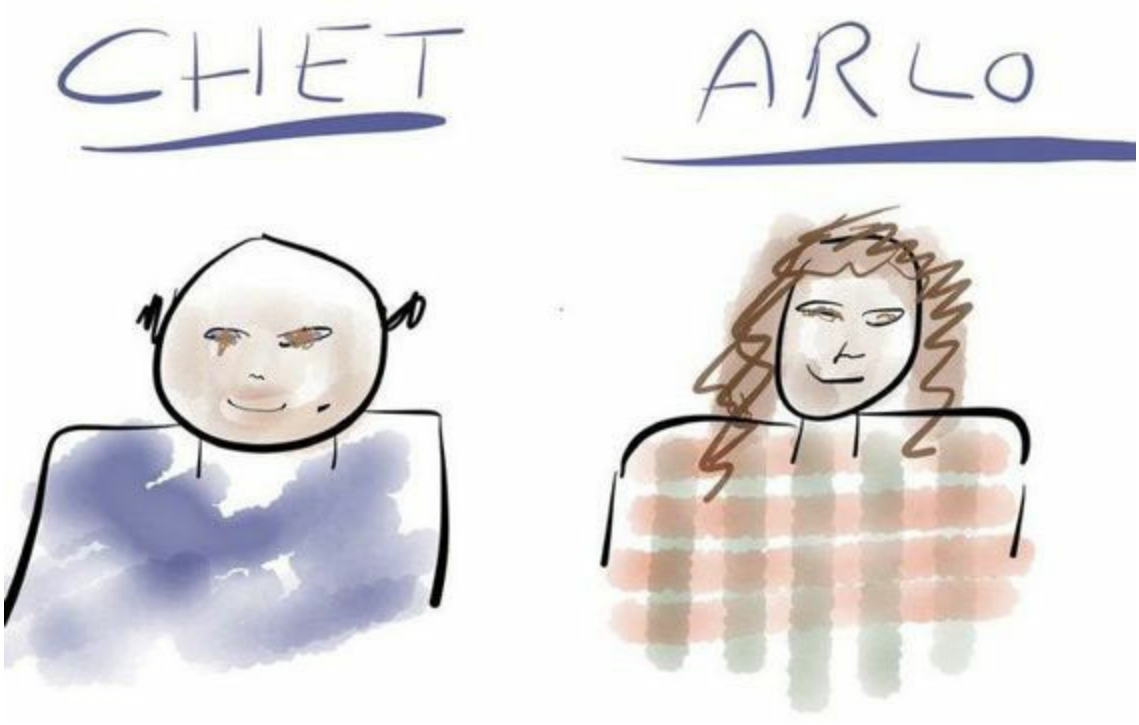
不幸的是，与这首歌所描述的相反，大公司总是能够得到它想要的东西——大量昂贵的培训。这些培训都以重量级的方法开展，同时被冠以大规模敏捷的名字。当然，大公司能够从中得到一些益处，不管怎么说，以任何形式关注改进总比不关注要好。而且，只要这些方法在一定程度上包含一些真正的敏捷思想，组织培训的公司就肯定能够受益。

在此，我想讨论的是滚石乐队的那首歌所说的，也就是你需要的东西。那么，若想全面应用敏捷思想，大公司都真正需要知道什么呢？

敏捷很简单，但要做到并不容易。

敏捷是很简单的。最流行的敏捷方法Scrum只包含三个角色、几个活动以及一个主要的工件，即测试过的可用软件。

然而，这并不意味着敏捷很容易。要确定什么样的产品是客户想要的依然很困难，同样困难的还有开发出符合需求的软件。可是敏捷真的很简单。可以说，简单就是敏捷的本质及精髓。既然敏捷是简单的，那么所谓的大规模敏捷又如何呢？



大规模敏捷必然也是简单的，否则它就不是敏捷。

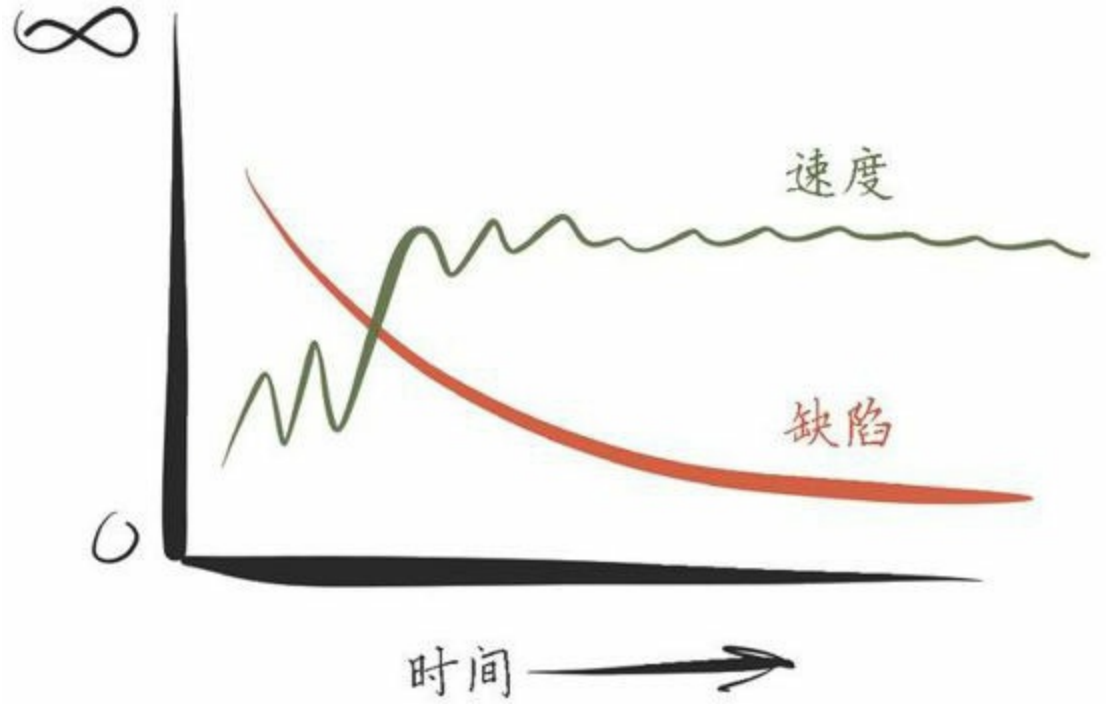
敏捷方法培训师Chet Hendrickson指出，既然敏捷是简单的，那么大规模敏捷应该也是简单的，甚至要更简单。否则，它就不再是敏捷。因此，对于那些自称“大规模敏捷”的复杂方法，我们一定要抱着高度怀疑的态度。

另一位敏捷方法培训师Arlo Belshee也认为，如果你的所有开发团队都已经很精通敏捷软件开发方法，那么大规模敏捷就不是什么问题。如果你的所有开发团队都能够熟练地划分功能特性，正确地选出他们能够在一次冲刺中（或者是在估算的时间内）完成的功能特性数，还能够按时交付没有缺陷的已集成软件，那么大规模敏捷同样应该很简单。此外，敏捷熟练度（Agile Fluency）这一概念的提出者Diana Larsen和James Shore也有类似的表述。

下面，我们就来探讨这一问题：敏捷是简单的（但并不容易），如果每个团队都能够真正地按照敏捷的方式进行软件开发，那么大规模敏捷容易吗？

如果你的团队已经达到了真正的敏捷……

敏捷团队每天都与业务人员一起工作（《敏捷宣言》之原则4）。他们每几周就会交付可用的软件（原则3）。他们以可用的软件来衡量自己（原则7），以可持续的方式进行工作（原则8），同时不断地关注优秀的技术和良好的设计（原则9）。

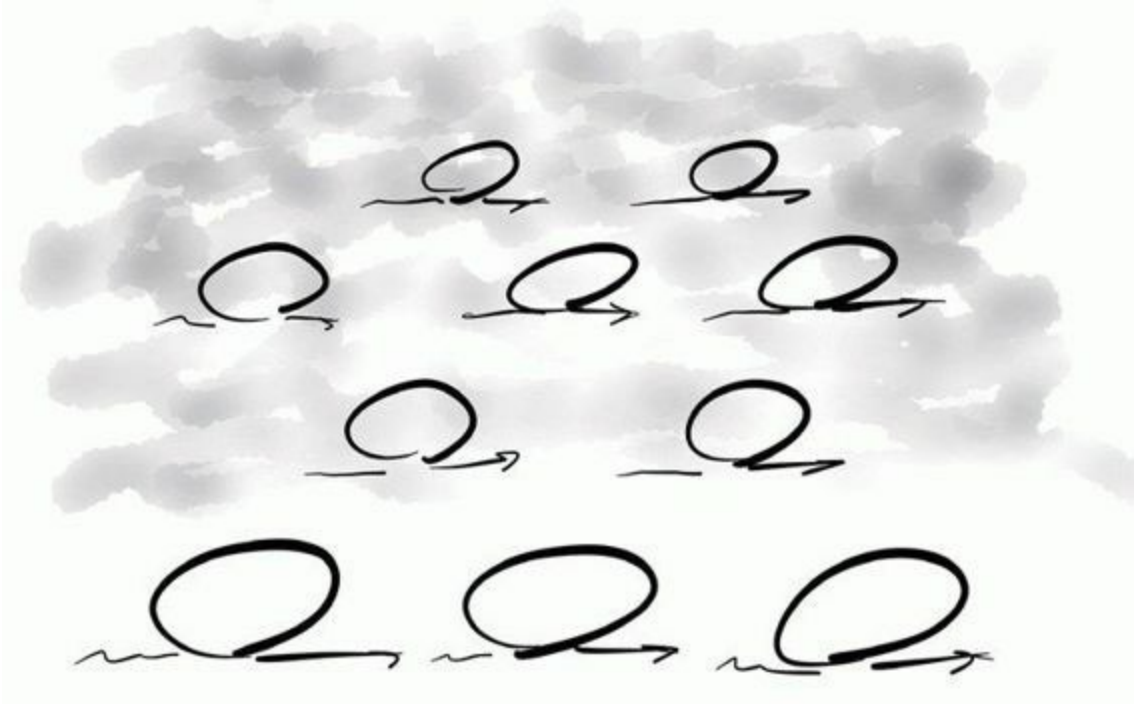


我说的是真正的敏捷……

除了在一开始有些不协调之外，熟练的敏捷团队能够以稳定的速度持续地开发功能特性，而且所交付的软件的缺陷数量要远远少于熟练掌握敏捷方法之前的数量。

熟练的敏捷团队，其敏捷熟练度是可见的。他们以稳定的、可预测的速度真正地完成工作。如果你的团队已经接近这一水平……

……那么你可能已经做到了大规模敏捷。

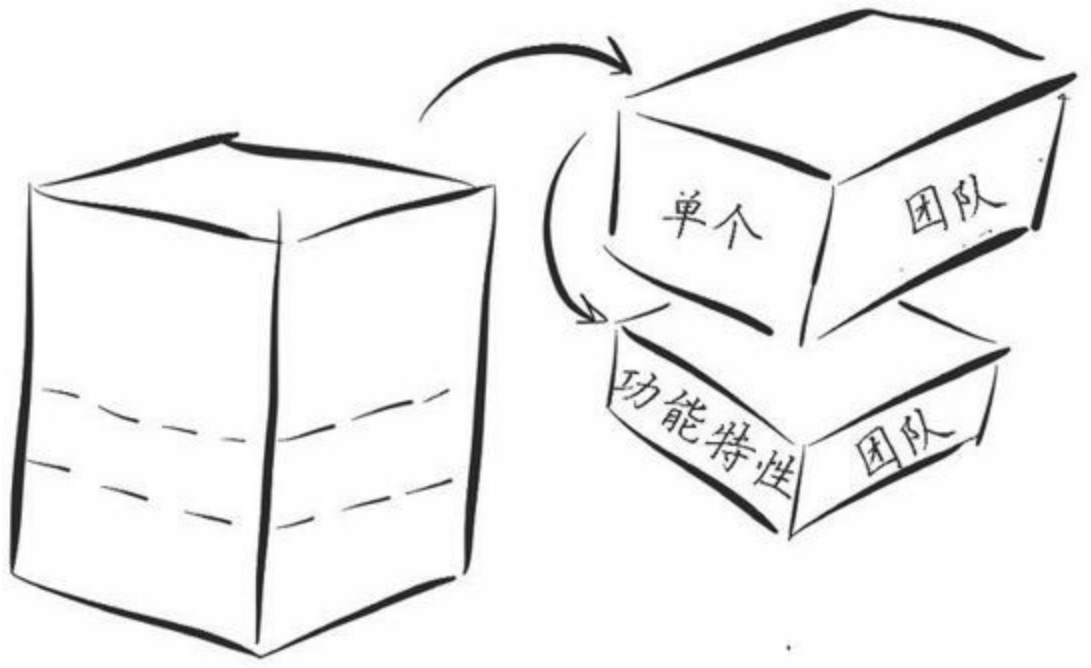


假定你的所有团队都达到了这样的水平：他们每天都能够与业务人员一起工作，由这些业务人员负责向开发团队描述软件需求；开发团队每两周就能够构建出可用的软件。

如果公司的所有软件都能够由单个敏捷团队来开发，那么该公司可能已经做到了大规模敏捷。

真的，你可以仔细想一想。如果待做的一切工作都能够由单个小团队完成，那么大规模敏捷就可以被认为是让每一个团队都去学习敏捷方法，同时安排业务人员将团队成员组织起来并对他们进行相应的指导。

大规模敏捷就这样完成了，除了基础工作之外并不需要额外的工作。当然，这些基础工作是很困难的，本书的其余章节已经对此进行了讨论。但可以看出，并不需要去做大的上线、转变以及最终的企业级敏捷之类的活动。

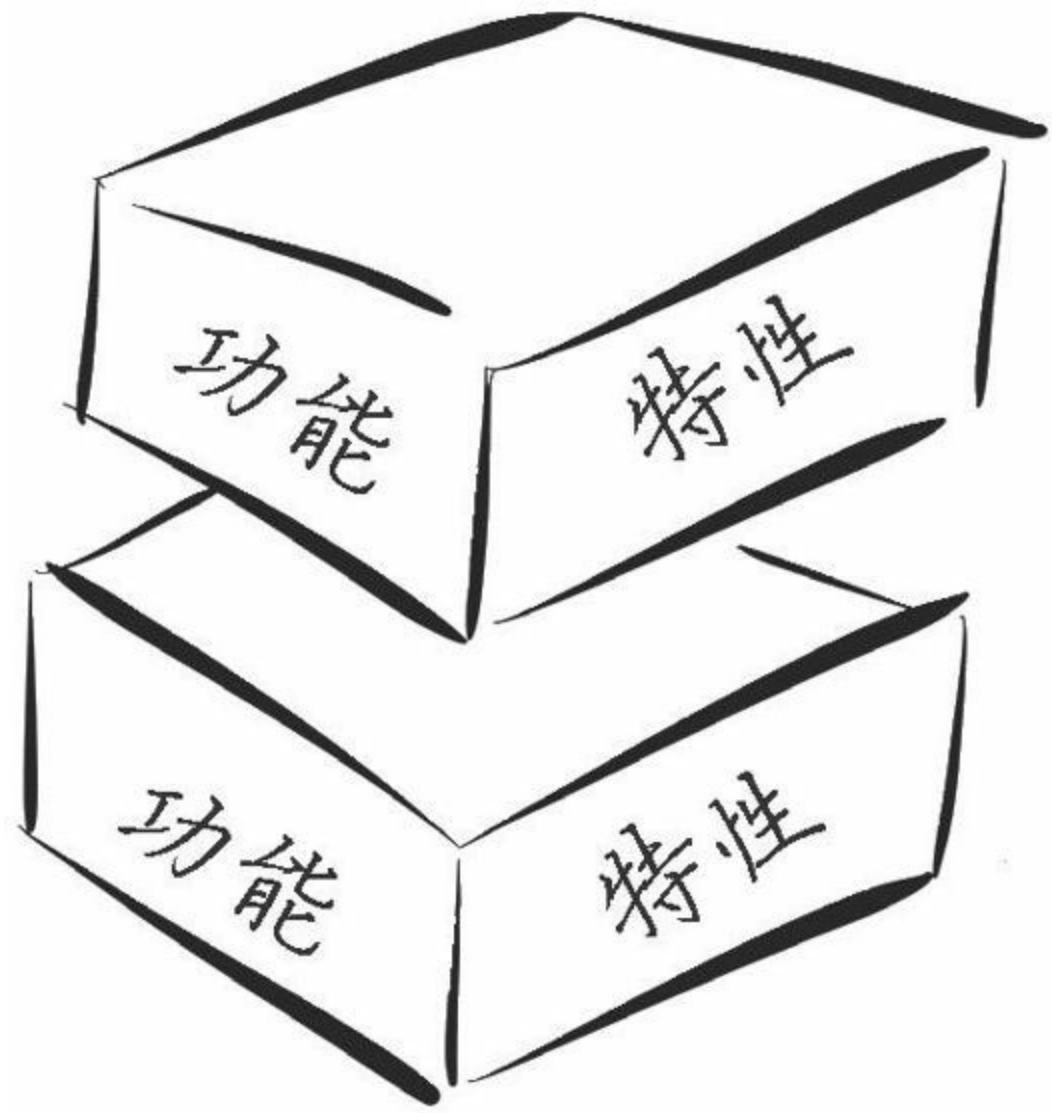


如果你想要的比单个团队能够做的要多，该怎么办？

真正的敏捷团队每几周就能够开发出多个功能特性。即使保持一个团队能够这样满负荷地工作也是不容易的：为了做到这一点，你必须要有很多关于产品的想法。但也有可能你要做的是一款很大的产品，如文字处理软件或者编辑照片的图像处理软件。你觉得有足够多的工作可以同时让多个团队都保持忙碌状态。

好吧，首先来验证这样做的必要性。先让单个团队以达到敏捷的要求开发这一产品，然后计算该团队交付功能特性的速度。有了这一数据之后，再看看是否真的需要更多的功能特性。极有可能并不需要，因为客户掌握新功能特性的速度要比团队交付功能特性的速度慢。但是我们仍然不能排除一种可能性：有足够多的工作可以让多个团队都忙碌起来。

哈哈！现在我们不得不采取大规模敏捷了，不是吗？



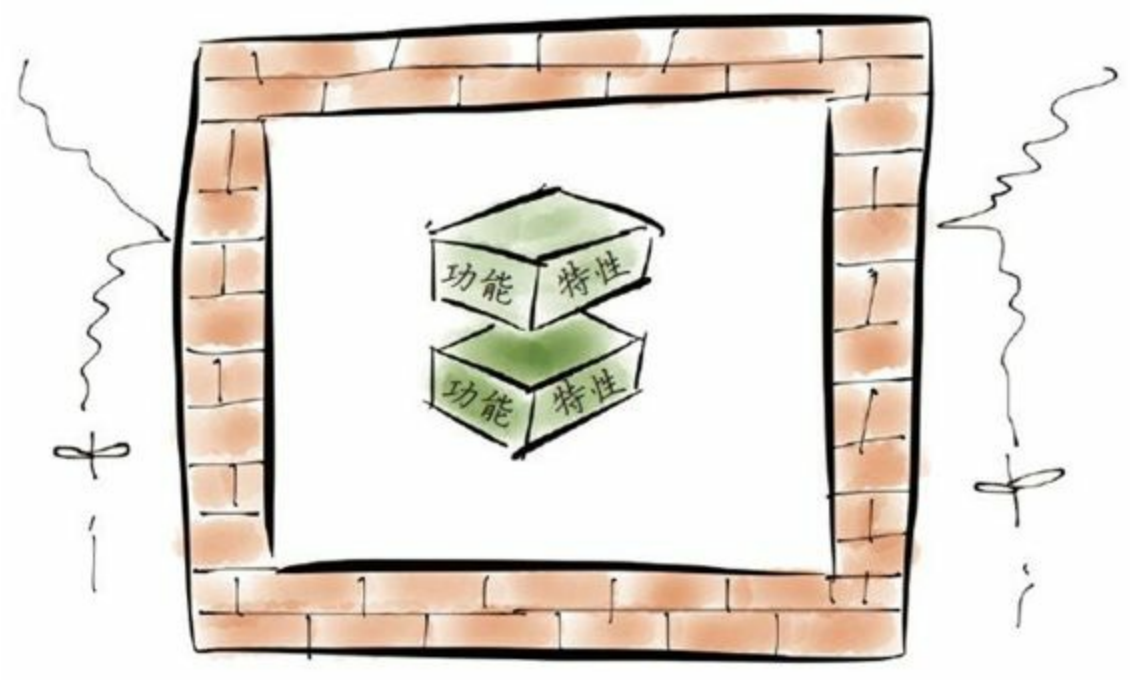
功能特性团队

不过，我们还是有可能不用这样做。早在上个世纪，人们就提出了“功能特性团队”这一概念。它是一个小团队，负责将功能特性添加到产品中。为了得到更多的功能特性，需要安排更多的功能特性团队，所有团队将开发出的功能特性加入同一产品中。那么，如何在单位时间内获得更多的功能特性？答案很简单，只需再增加一个功能特性团队即可。

这种方式并没有过多涉及大规模吧？如果每个团队都能够像真正的敏捷团队那样知道该怎么做，你只需增加功能特性团队的数量即可。这样一来，任何由功能特性组

成的产品都能够以你想要的速度被构建出来。

难道我们没有忽略什么吗？这些团队要怎样协调配合？既然多个团队同时构建功能特性，那么他们怎样才能避免相互影响？



敏捷团队通过测试进行协调配合。

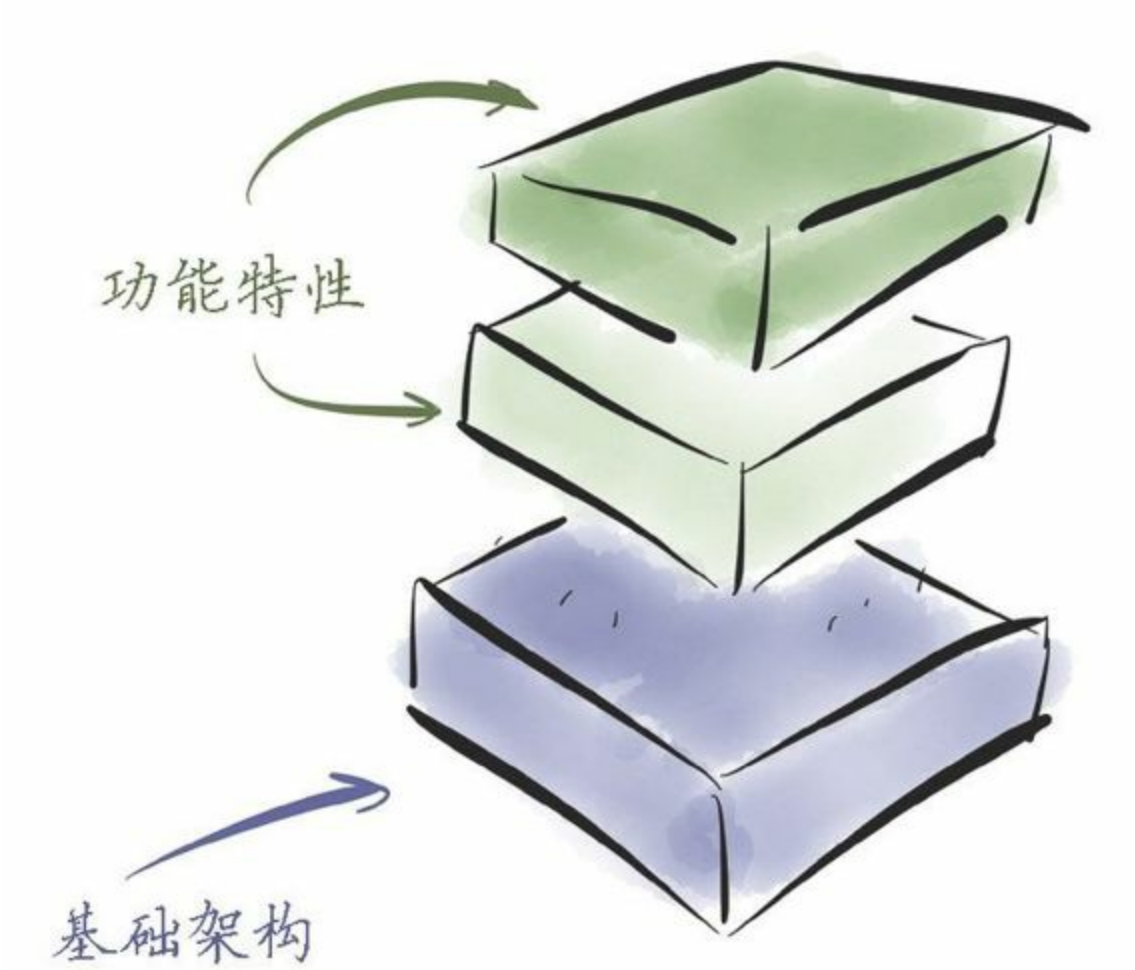
我们知道，敏捷团队每两周就能够完成相当数量的小的功能特性。在为期两周的迭代期间，一个团队就能够轻松地完成15个甚至20个这样的功能特性。多个团队怎样做到互不妨碍呢？

事实证明，这很简单。熟练的敏捷团队通过构建不断增大的自动化测试库就可以做到互不妨碍，这里所说的自动化测试库是用验收测试驱动开发以及测试驱动开发方法构建的。这些测试不仅能够帮助敏捷团队知道他们何时完成了某个功能特性，而且还可以作为不断增长的回归测试集，以保证已经完成的所有功能特性能够一直正常。

这一方法同样适用于同时工作的多个功能特性团队。每个团队每次构建新的小功能特性时，就向公共代码库中增加该功能特性以及相应的自动化测试代码。所有团队每天都这样做，正如只有一个团队时一样。他们会一直运行所有测试。如果有时候某个团队试着迁入代码时发现测试失败，那么他们需要在正式迁入代码之前修复这一问题，从而保证当前的代码库始终运行所有测试。

不同的团队所做的工作有可能会发生冲突吗？的确有可能，如果真的发生了冲突，那么所涉及的团队可以一起查找原因。不过，解决这一问题的通用方法很简单：如果在加入修改的内容之前测试可以运行，而在加入之后测试不能够运行，那就说明修改的内容破坏了其他代码。这时，需要将修改的内容找出来，然后修复它，这样所有的测试就可以继续运行了——包括新加入的测试以及之前的测试。

敏捷团队当然会这样做，他们会学着发布越来越小的版本。当发布小版本时，破坏其他代码的可能性就会变得很小。当真的发生这样（罕见）的情况时，很容易就能找出问题，因为团队只是增加或者修改了少量代码。



好吧，功能特性团队是这样的。那么基础架构呢？

如果产品足够大，且需要由多个功能特性团队共同完成，那么这些团队会依赖同一基础架构。对于基础架构的改变又要如何处理呢？

答案是，使用同样的方法。敏捷团队会根据需要每几周就自由地改变基础架构，而且这些改变都有自动化测试的支持。功能特性团队同样可以这样做，每个团队根据自己的需要对基础架构进行相应的改变，同时增加相应的自动化测试，然后再频繁地迁入代码。

那么，有必要组建专门的基础架构团队吗？如果团队已经熟练地掌握了敏捷方法，那么很多时候并不需要这样做。在采用敏捷方法的情况下，专门的团队经常会无用

武之地。但如果你坚持选择组建一个这样的团队，而且是敏捷团队，那么他们仍然可以顺利地对基础架构进行改变以同时满足多个团队的需要，而且这些改变都有自动化测试的支持。不过，我还是建议让功能特性团队负责处理基础架构的改变，让他们自己根据需要进行协调。即使你不采纳我的建议，坚持选择组建一个专门负责基础架构的团队，也没有必要刻意实现大规模敏捷。

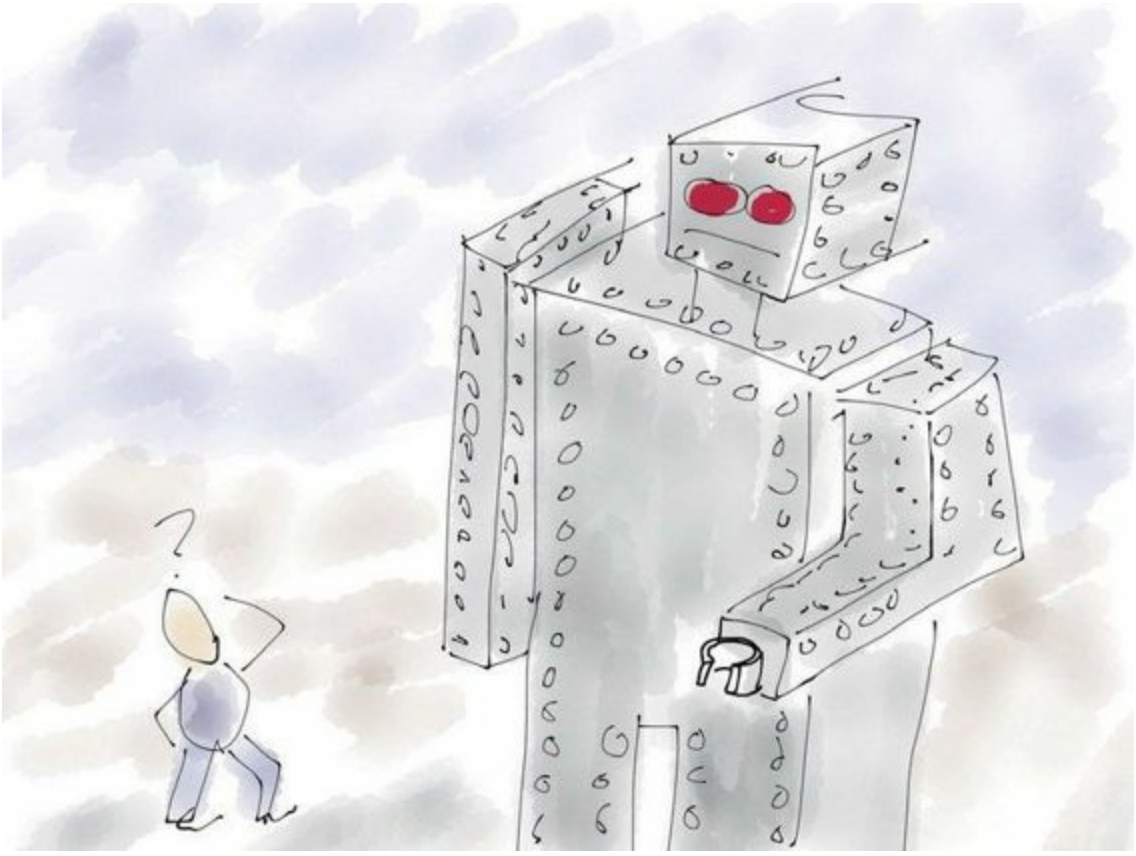
不要忘了，如果你的每个团队都能做到敏捷，那么很有可能你根本就不需要功能特性团队。即使你需要功能特性团队，也不需要组建专门的基础架构团队——你只需要给多个功能特性团队授权，让他们自己根据需要进行协调。

到目前为止，一切都很好。

那些所有工作都能够由单个团队完成的公司，并不需要做特别的事情来实现大规模敏捷。而那些所需功能特性的数量超过单个团队的开发能力的公司，则可以组建功

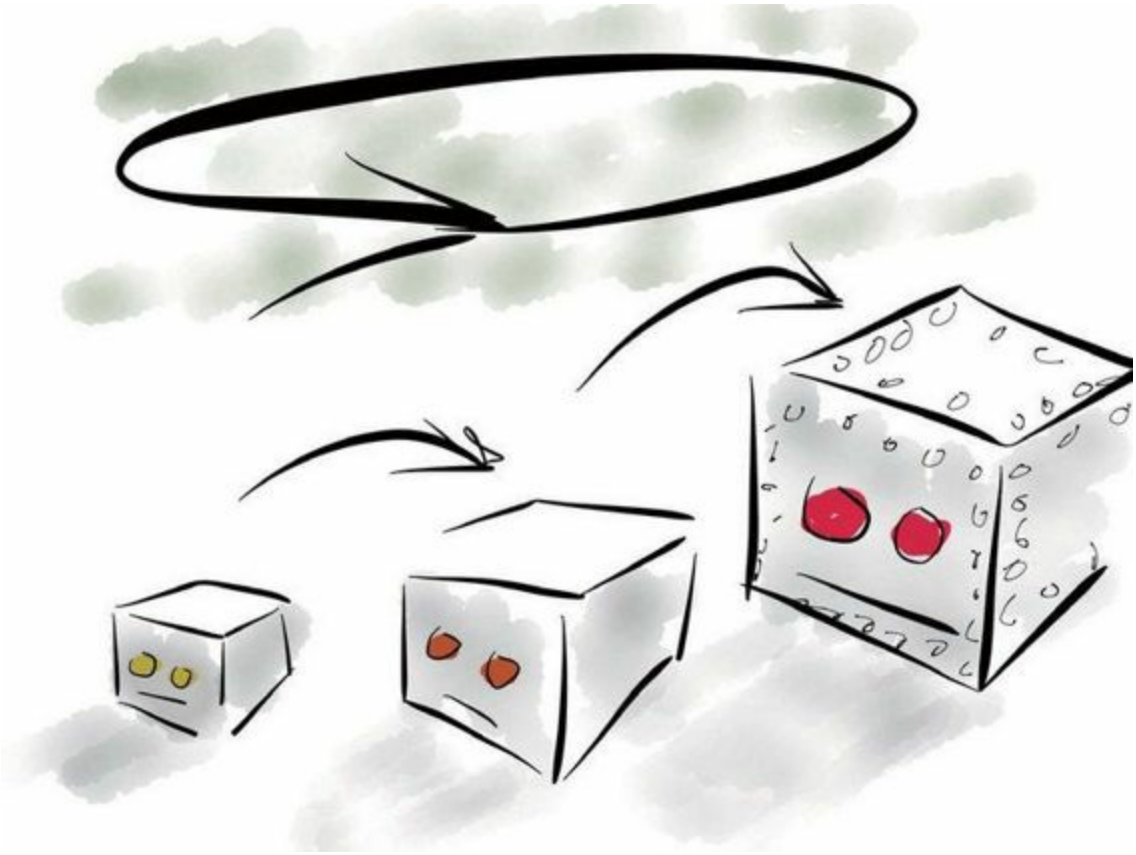
能特性团队，并且同样不需要做任何其他的事情来实现大规模敏捷。

在我所见过的大多数公司中，大部分任务都是由单个团队完成的。而在少数公司中，我看到的则是集成度高、任务量大的产品。它们可能需要有多个功能特性团队。还有其他情况吗？



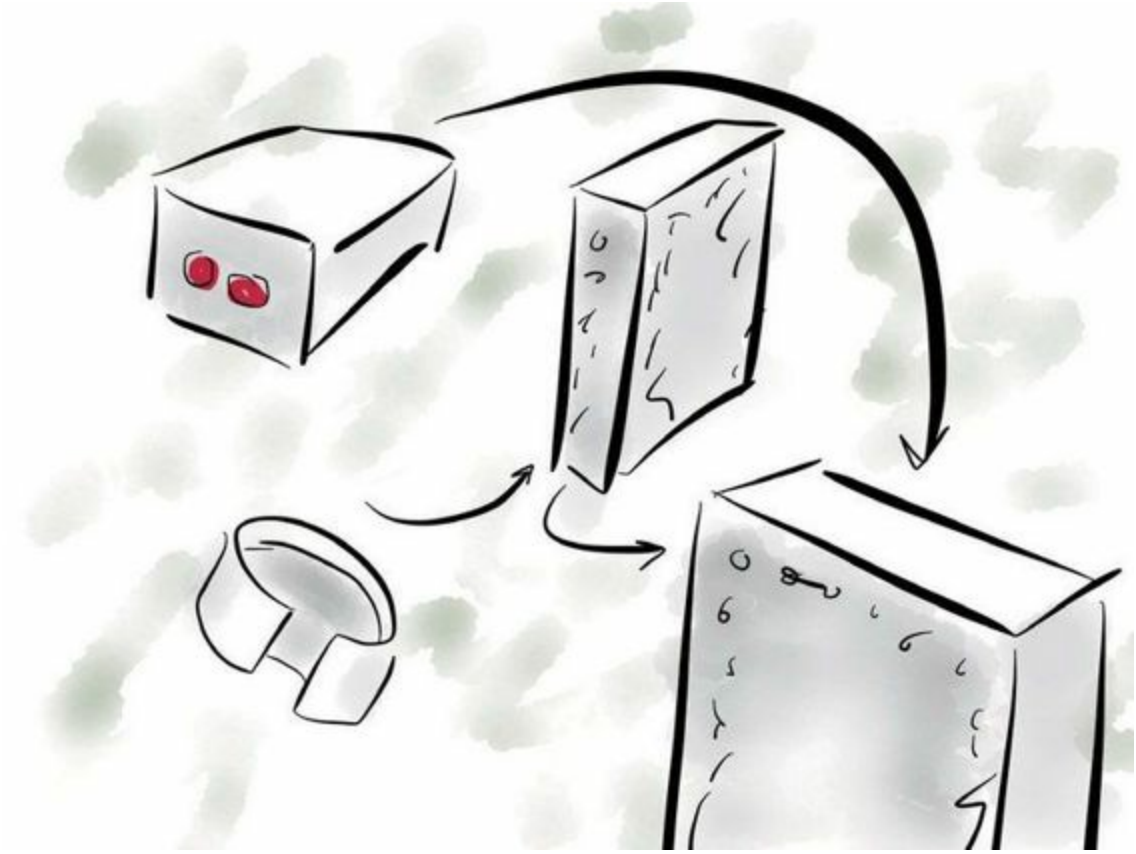
庞大的项目

还有一些公司从事真正庞大的项目，有几百个甚至几千个开发人员共同完成一件事情。如果你不是在这样的公司里工作，或许只需让你的每个团队都能够按照敏捷的方式工作即可。你现在就可以合上本书，或者直接跳到下一章的结论。但是，对于这样庞大的项目，你很有可能会好奇应该怎样做。



首先，让庞大的项目逐渐增大。

如果你正要启动庞大的项目，即使是在已有的基础上构建，标准的敏捷方法依然会发挥作用。从单个团队开始，使它逐渐变大。随着项目的进行，构建基础架构并对它进行扩展。如果需要，还可以增加功能特性团队。



然后，对庞大的项目进行分割。

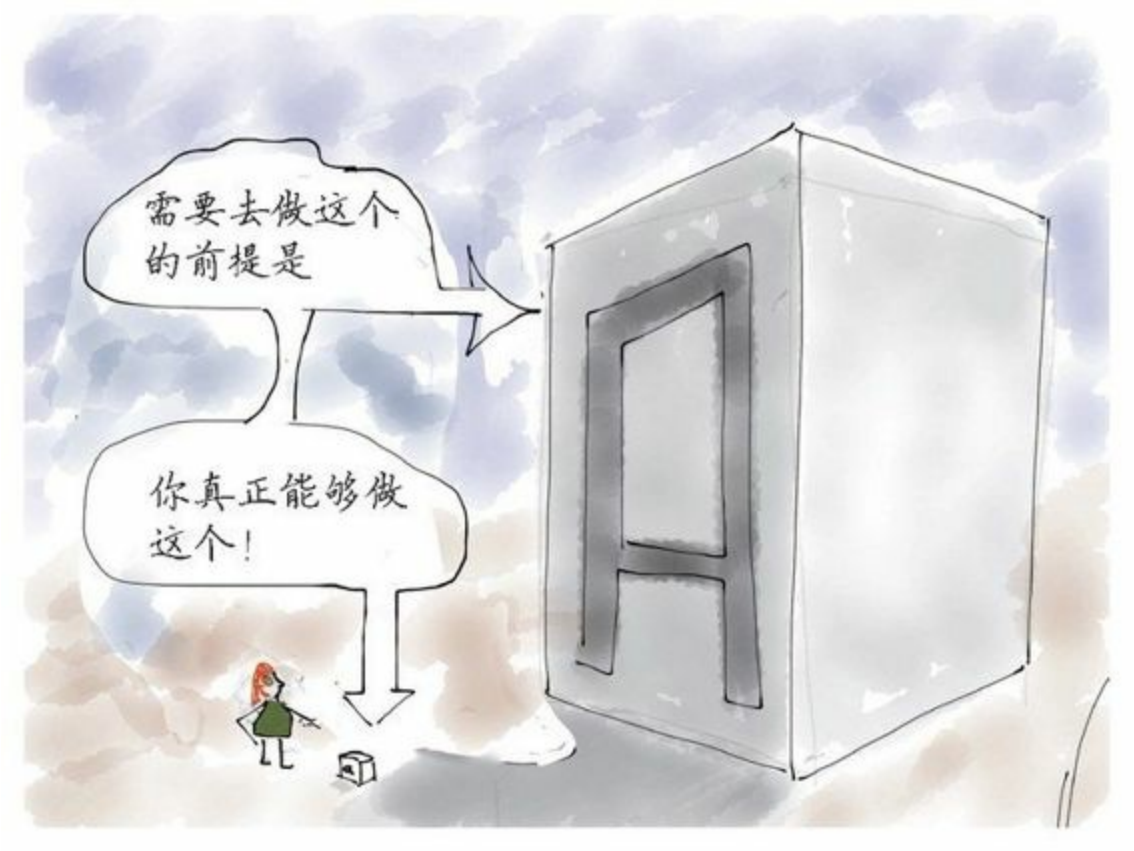
事实证明，即使是庞大的项目，几乎所有工作也都是由单个团队完成的。我们已经知道怎样做了：按照标准的敏捷方法去做即可。

即使是庞大的项目，也可以通过增加团队的数量来提高速度。以功能特性团队的标准管理新增的团队，并且只需采用标准的敏捷方法即可。

那么，还剩下什么呢？需要不止一个团队去完成，并且不能再被分割为采用敏捷方法完成的小任务，这样的项目是否真的存在？

大多数情况下，我很怀疑它是否存在。我认为，没有什么大项目真正是不可分割的。如果的确有的话，不管是否采用敏捷方法，都没有人知道应该怎样做。安排很多人去做同一个项目，其本质就是分配工作。如果不知道如何分配工作，那么即使增加人手也无济于事。

若知道如何分配工作，那么大部分的工作几乎总是可以用标准的敏捷方法去做。还存在什么充分的理由必须要用复杂的方法实现大规模敏捷吗？或许有吧。不过我的建议是，不妨先等一等、看一看。



本章要旨

当团队不能以敏捷的方式工作时，公司显然还没有做好“转变”或者采取“大规模”敏捷的准备。你不会想要向做不到的事情转变，也不会采取行不通的方法。

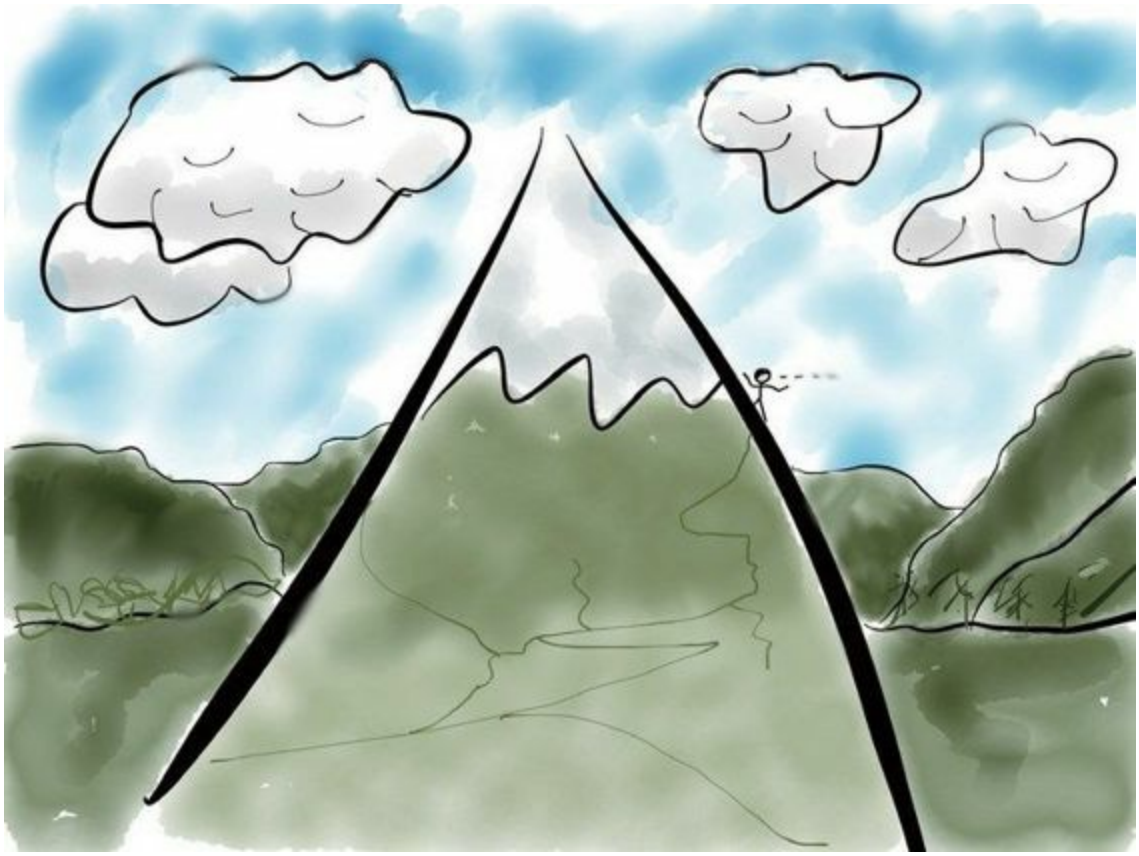
在这样的情况下，你需要做以下三件事。

第一，组建可以按照敏捷方式工作的团队。

第二，将公司所能够提供的最重要、最有价值的工作交给敏捷团队，然后不再参与，让他们自主发挥。

第三，根据功能特性继续组建敏捷团队。你可能会发现并不需要大规模敏捷，而更有可能只需要敏捷即可。

第22章 结论



祝贺你终于读到这一页！为了帮助你记住本书所述内容，本章将总结你在书中所经历的事情。

设想你正在攀登一座名为“软件开发”的山峰。你可能是一位新手，还在山脚附近，正沿着陡峭的山路往上走，偶尔还需要攀爬岩石；你也可能已经是老手，有一套完整的登山工具以及相当丰富的登山知识，甚至能够向其他登山者提供帮助；你还有可能已经成为令人称赞的徒手攀岩者，只需一只手抠住缝隙就能继续向上爬，而且脚能踢过头顶并在玻璃墙一般的峭壁上找到下一个缝隙。

无论你有什么样的水平，如果你像我一样，那么肯定会花很多时间研究你面前的这座山峰。你会花不少时间思考下一步应该怎样向上攀登，以及依靠哪几块肌肉的力量迈出这一步。

本书是与你同登一座山峰的另一个人所带给你的话语与图片。他找到了一个相当平坦舒适的地方，可以坐下来环顾四周。他向山外望去，瞧见异常漂亮的风景。这风景位于登山者的身后，因而几乎总是处于他们的视野之外。他又向下看去，瞧见山峰的轮廓，同时还发现其他登山者，其中有些人的登山水平不如他，有些人则比他强。此外，他还瞧见很多条上山的路，因此能够辨别走哪条路容易、哪条路困难、哪条路安全、哪条路危险。

然后，他又向上望去，瞧见笼罩着山顶的云层和雾霭。他意识到向上攀登的路还有很长，比已经登过的路程还要长很多。他还瞧见一些看似可走的路，并找到一些可供停下来环顾四周的有趣之处。

他拍下所看到的一些风景，同时写下自己的一些想法和发现。此外，他也勾画出一些路，想以此告诉你识别好路的经验，以及当所选之路被证明不好时他所采取的行动。他愿意将所有这些都告诉你，以此丰富你的登山之旅，同时提醒你不要忘了欣赏那些美丽的风景。正是这些美丽的风景让你觉得无论登山有多困难，都是值得的。他还想提醒你，有时候登山的价值就在登山本身，登得越多，水平就越高。

他最后说道：“这就是我对软件开发的理解。对此，你又是怎样理解的呢？”

感谢你阅读本书！